VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Pokročilá konfigurace jádra GNU/Linux prostřednictvím NetLink Socket

Advanced GNU/Linux Kernel Configuration based on NetLink Socket

2013 Jan Hellstein

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jan Hellstein**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Pokročilá konfigurace jádra GNU/Linux prostřednictvím NetLink Socket
Advanced GNU/Linux Kernel Configuration based on NetLink Socket

Zásady pro vypracování:

Cílem práce je průzkum možností pokročilé konfigurace zejména síťového subsystému jádra operačního systému GNU/Linux prostřednictvím NetLink Socket. Práce se zaměří především na možnosti sestavování GRE a L2TPv3 tunelů pro účely tunelování L2 protokolů v směrovaném prostředí.

1. Seznamte se s možnostmi síťového subsystému jádra OS Linux. Zaměřte se na možnosti jeho konfigurace prostřednictvím NetLink Socket.
2. Nalezněte a důkladně zdokumentujte způsoby vytváření IPoGRE, ETHERNEToGRE a L2TPv3 tunelů s pomocí NetLink. Dále se zaměřte na související konfigurace IP a MAC adres na existující rozhraní a manipulaci se směrovací tabulkou jádra OS.
3. Nalezený způsob konfigurace implementujte ve zvoleném programovacím jazyce.
4. Navrhněte a realizujte testovací síťovou topologii, realizujte ji a výsledné řešení důkladně otestujte. Při testování srovnejte zátěž systému a propustnost tunelů jednotlivých typů enkapsulací.
5. Zhodnoťte dosažené výsledky a sumarizujte vhodnost použití jednotlivých technologií pro typické scénáře.

Seznam doporučené odborné literatury:

BENVENUTI, Christian. Understanding Linux network internals. USA: O Reilly Media, Inc., 2006, 1035 s. ISBN 9780596002558.
BECK, Michael. Linux kernel internals. 2nd. USA: Addison-Wesley, 1998, 480 s. ISBN 9780201331431.
SMITH, Roderick W. Advanced Linux networking. USA: Addison-Wesley, 2002, 752 s. ISBN 9780201774238.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Martin Milata**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2013 .....

Rád bych na tomto místě poděkoval svému vedoucímu, Ing. Martinu Milatovi, za jeho rady a připomínky. Dále bych chtěl poděkovat všem ostatním, kteří mi během psaní práce pomohli, protože bez nich by tato práce nevznikla.

Abstrakt

Tato práce si klade za cíl vysvětlit způsob manipulace se síťovou částí jádra (kernelu) GNU/Linux pomocí Netlink socket. Jádra operačních systémů přímo komunikují s hardwarem počítače, a pokud chce uživatel změnit či nastavit například IP adresu na určitém síťovém rozhraní, provede to většinou nějakým specializovaným nástrojem. Tyto nástroje pak musí toto nastavení nějakým způsobem předat jádru OS. Netlink socket je jedním ze způsobu tohoto předávání dat mezi user-space a kernelem. Mým úkolem v této práci bylo napsat programy, které pomocí Netlink socket vytvoří síťové tunely, a toto řešení zdokumentovat a otestovat.

Klíčová slova: GNU/Linux; Linux; kernel; Netlink socket; síť; tunely; GRE

Abstract

This thesis is focused on explaining how we can use Netlink socket to configure the GNU/Linux kernel's networking subsystem. It is kernel that directly communicates with computer hardware, so when user wants to change something, e.g. an IP address on some interface, s/he will probably use some special utility. These utilities then use a certain method to pass these new configurations to kernel. Netlink socket is one of these methods. My goal in this thesis was to write a program(s) that use Netlink socket and are able to create network tunnels. Founded solution is afterwards documented and tested.

Keywords: GNU/Linux; Linux; kernel; Netlink socket; network; tunnels; GRE

Seznam použitých zkratek a symbolů

ACK	–	Acknowledgement
API	–	Application Programming Interface
ARP	–	Address Resolution Protocol
CPU	–	Central processing unit
IP	–	Internet Protocol
IPSec	–	IP Security
MAC	–	Media Access Control
MTU	–	Maximum transmission unit
OS	–	Operating System
TCP	–	Transmission Control Protocol
TCP/IP	–	Transmission Control Protocol / Internet Protocol
TTL	–	Time to live
UDP	–	User Datagram Protocol
VLAN	–	Virtual LAN
VPN	–	Virtual Private Network

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Tato diplomová práce je členěna do několika kapitol, které odpovídají postupu řešení daného problému, tedy vytváření tunelů pomocí Netlink socket.

V navazující kapitole přiblížím historii vzniku UNIXu a některé z jeho obecných znaků. Dále vysvětlím, jaký má k němu vztah Linux a jaký je rozdíl mezi Linuxem a GNU/Linuxem.

Poté se budu více věnovat Linuxu, resp. jeho vnitřní struktuře. Ukáži, že Linux může být logicky rozdělen do několika oddělených částí, přičemž jednu z těchto částí budu později konfigurovat.

Popíši některé způsoby, kterými je možno komunikovat mezi Linuxem a user-space, a zároveň vysvětlím, proč je tato komunikace potřebná.

Netlink socket je jedním ze způsobů této komunikace, navazující kapitola se proto bude věnovat jeho teoretickému popisu. Ze začátku půjde o popis obecný, tedy k čemu přesně slouží a jaké má vlastnosti. Dále ukáži, jak se vytváří socket, kterým bude Netlink zpráva odeslána, a poté přiblížím, jak vypadá Netlink hlavička a jaké možnosti skýtá. Stejně jako Netlink hlavičku popíši i některé RTNetlink hlavičky, ty, které budu potřebovat ke splnění úkolu. Nakonec se podívám na způsob přidávání atributů za RTNetlink hlavičky, na některé jejich typy a na to, k čemu slouží.

Poté se dostávám k vyústění práce, tedy tvorbě tunelů. Ukáži úseky programů, které vycházejí z nabytého teoretického základu a které jsou schopny pomocí Netlink socket vytvořit funkční tunel mezi dvěma počítači.

Vytvořený tunel pak otestuji na jednoduché síťové topologii a změřím, jaký dopad mají enkapsulace na vytížení CPU při odesílání velkého množství dat vytvořeným tunelem.

Poslední kapitolou je Závěr, ve kterém shrnu dosažené výsledky mé práce.

2 UNIX, GNU a Linux

Na začátek bych rád objasnil tyto pojmy, protože si je lidé velmi často pletou (především Linux a GNU/Linux), a může tak docházet k nepochopení.

2.1 UNIX

Práce na UNIXu začaly v roce 1969 v Bell Labs, což bylo výzkumné centrum americké telefonní společnosti AT&T, tehdy monopolisty ve svém oboru. [1] Systém vznikl s cílem umožnit několika programátorům najednou přístup ke zdrojům počítače, čili umožnit jim na jednom počítači pracovat současně. [2]

První verze UNIXu vyšla v roce 1971 , tehdy ještě psaná v jazyce B Kena Thompsona, (jazyce symbolických adres) [3]; čtvrtá verze (1973) byla přepsána do jazyka C [4], který v Bell Labs vytvořil Dennis Ritchie, jeden z autorů UNIXu. [5] Díky tomu se systém stal snadno přenositelným na jiné platformy.

Vývoj UNIXu pokračoval a postupně se UNIX rozšířil natolik, že se z jeho jména stala ochranná známka Bell Labs. [6] Tuto ochrannou známku nyní vlastní The Open Group. [7]

Některé z vlastností UNIXu:

- multitasking,
- víceuživatelský,
- přenositelný.

UNIX se skládá ze tří hlavních částí: [8]

- kernel,
- shell,
- programy.

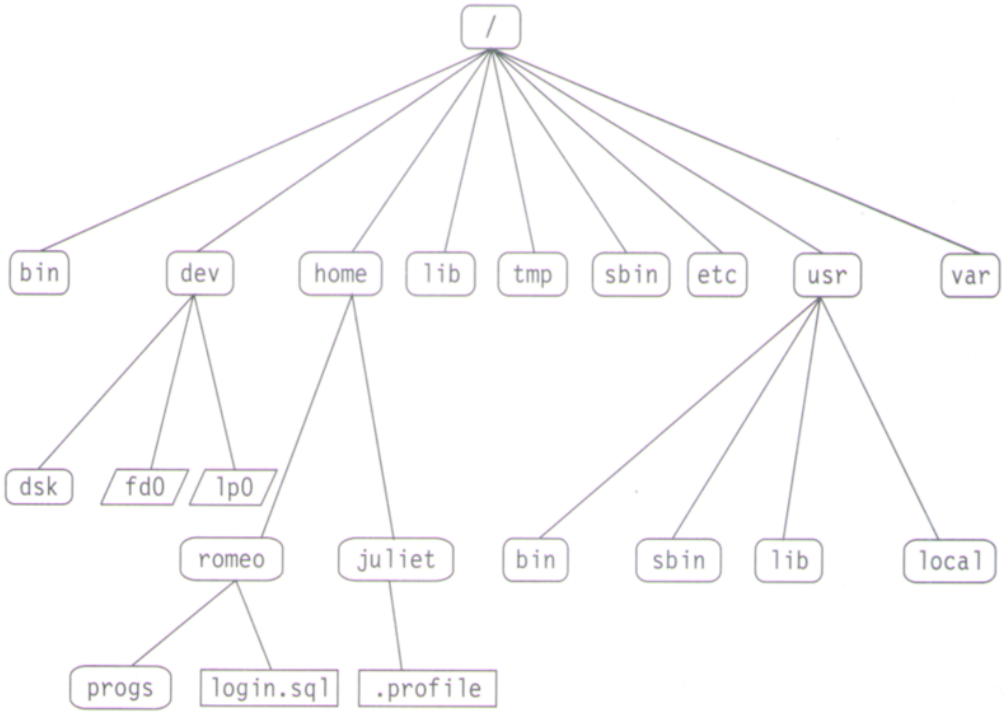
Kernel je jádro systému, které ovládá hardware (alokaci paměti, přístup na disk..) a shell je rozhraní mezi kernelem a programy (v dalším textu někdy jako user-space), které interpretuje příkazy napsané uživatelem.

UNIX dále zavádí následující vlastnosti/techniky: [8]

- vše je soubor nebo proces
  - proces je spouštěný program, který je identifikován svým PID (Program ID),
  - i složky jsou soubor (speciální typ souboru obsahující seznam souborů, které daná složka obsahuje) i vstupně/výstupní zdroje (hardware) jsou soubory. UNIX zde v podstatě zavádí abstrakci všech těchto zařízení do formy souboru, díky které můžeme použít stejné příkazy (read/write) na dokument nebo třeba

na připojené síťové zařízení. A soubor nakonec není nic jiného než sekvence bajtů; [9] [10]

- soubory i složky mají hierarchickou strukturu, kde nejvýše je root (/), viz obrázek 1, [11]
- konfigurační soubory jsou prostými textovými soubory, takže se dají konfigurovat bez nutnosti použití speciálních nástrojů. [12]



Obrázek 1: Struktura adresářů v UNIXu

Šestá verze UNIXu (1975) byla první verzí, která byla k dispozici i mimo Bell Labs. Na univerzitě v Berkeley v Kalifornii pak z této verze vzniklo BSD, Berkeley Software Distribution. Vývoj BSD přinesl několik inovací. Mezi nejdůležitější patří: [13]

- virtualizace paměti,
- TCP/IP protokol.

Mezi dnes používané systémy odvozené z BSD patří např. OpenBSD, FreeBSD nebo NetBSD. Na kódu BSD je postaven i Mac OS X. [14]

Mezi systémy odvozené z UNIXu patří např. HP-UX nebo Solaris. [15] [16]

2.2 GNU

GNU je rekurzivní zkratkou pro GNU's Not Unix. Jedná se o operační systém, jehož vývoj byl Richardem Stallmanem oznámen v roce 1983 jako reakce na stále větší množství proprietárního softwaru. [17]

Cílem projektu bylo vytvořit kompletní OS, od aplikací po kernel, který by byl *free* a umožňoval uživatelům číst a měnit zdrojové kódy a programy dále vylepšovat. Projekt dostal jméno GNU Project.

Nový OS měl být kompatibilní s UNIXem, protože:

1. se osvědčil jeho návrh (viz kapitola 2.1) a byl přenositelný,
2. budoucí kompatibilita usnadní přechod z UNIXu.

Mezi programy nového OS měly patřit kompilátory, textové editory, e-mailový klient, GUI, hry a mnohé další. Práce pokračovaly až do roku 1990, kdy už byla většina aplikací napsána, ale chyběl kernel. [18]

Kernel GNU Projectu, GNU Hurd, dosud nedosáhl stabilní verze. [19]

Z GNU vzešlo spousta doteď používaných aplikací; jejich seznam je umístěn na webu GNU. [20]

Mezi nejvýznamější patří např. GNOME, GCC (the GNU Compiler Collection) nebo bash (Bourne Again Shell). [21]

2.3 Linux

Vývoj Linuxu oznámil Linus Torvalds v roce 1991, přičemž jeho cílem bylo vytvořit svobodný a jednoduchý operační systém pro počítače i386, jenž by byl podobný licencovanému (nesvobodnému) Minixu, který byl variantou UNIXu. [22] [23]

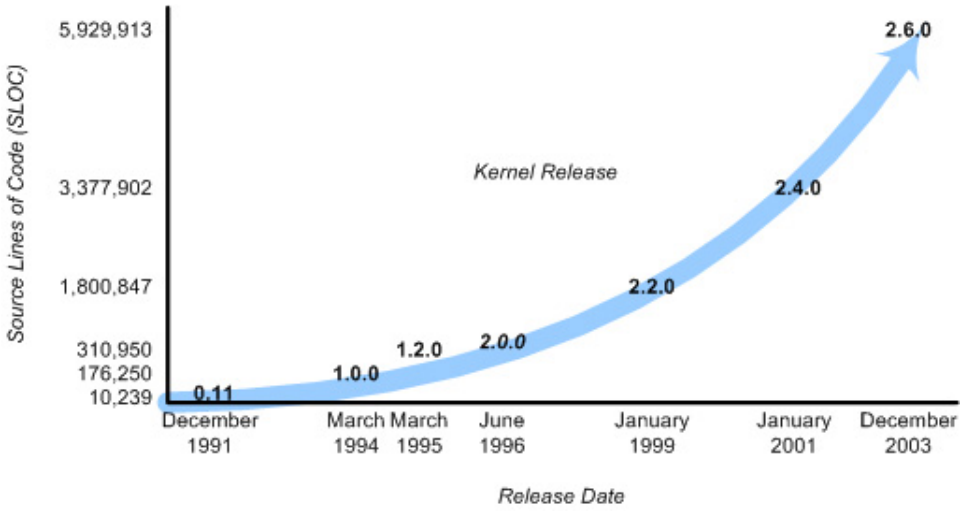
Verze 0.12 byla jako první vydána pod licencí GNU GPL. Tato licence uživatelům umožňuje volný přístup ke zdrojovým kódům, možnost jejich úpravy a následné další distribuce. [24]

Na Linuxu tak pracovalo postupem času stále více vývojářů a jeho vývoj do roku 2003 je zobrazen na obrázku 2, kde osa y znázorňuje počet řádků kódu. [25]

Linux verze 3.2 už obsahoval přes 15 miliónů řádků kódu; začátkem dubna 2013 byla aktuální verzí verze 3.8.5. [29]

2.3.1 Programovací jazyky

Linux vychází z UNIXu a je napsán ve stejném jazyce (C), který je de facto standardem pro vývoj kernelu. Kromě jazyka C se v Linuxu vyskytuje ještě assembler, který se používá u částí kernelu, jež jsou příliš závislé na hardwaru, například v kódu pro správu virtuální paměti. [30]



Obrázek 2: Vývoj počtu řádků kódu Linuxu

Při programování jsem tedy používal jazyk C a programy jsem psal v editoru Geany. Používám rolling release distribuci Arch Linux, programy jsem testoval na ní vždy s aktuální verzí Linuxu, z nichž poslední je 3.8.6-1-ARCH.

2.4 GNU/Linux

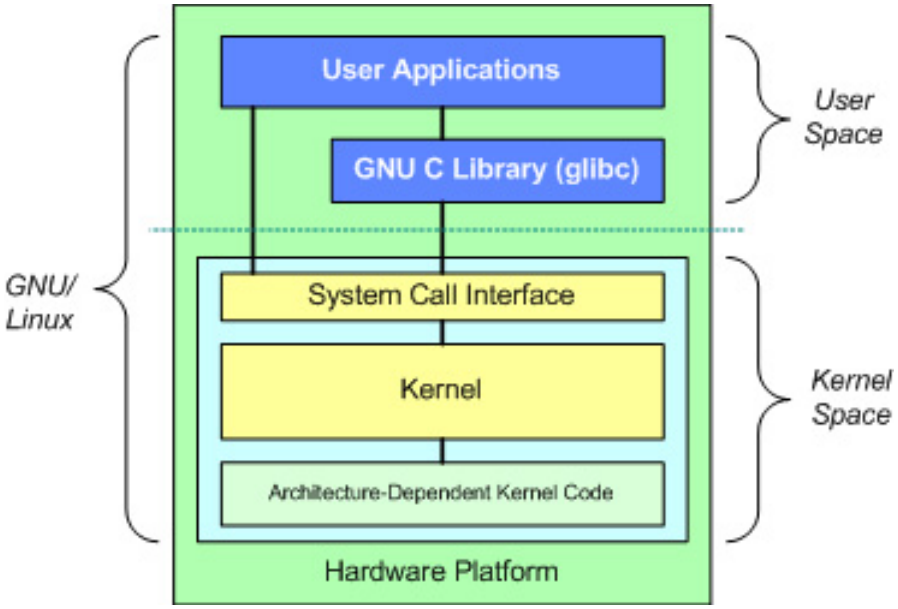
Jak je zmíněno výše, GNU Project měl na začátku 90. let vytvořeny aplikace, ale neměl kernel; Linux byl kernel, ale bez aplikací. Kombinací obou vzniká GNU/Linux, kompletní operační systém. Stejně jako BSD vychází GNU/Linux z UNIXu a adoptuje jeho design i vlastnosti (viz kapitola 2.1). Tyto a další podobné systémy se pak nazývají UNIX-like neboli unixové systémy. [31]

3 Linux

Linux je tedy jádro operačního systému GNU/Linux. Kernel je pak obecným označením pro jádro operačních systémů, nicméně v této práci budu význam slov Linux a kernel považovat pro zjednodušení za ekvivalentní.

Informace a obrázky v této kapitole jsou čerpány z [25], [35], [26] a [27], pokud není uvedeno jinak.

Základní architekturu GNU/Linux ukazuje obrázek 3. V user-space jsou spouštěny aplikace a nachází se zde i GNU C Library, jež poskytuje funkce pro použití systémových volání – kernel a user-space obsazují oddělené chráněné adresní prostory, systémová volání slouží k překročení této hranice a umožňují aplikacím využít služeb, které kernel poskytuje.



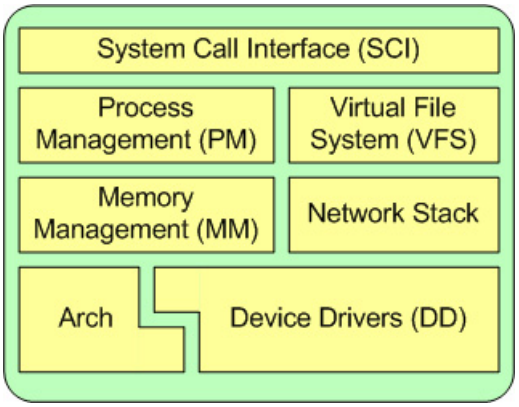
Obrázek 3: Architektura GNU/Linux

Mezi systémová volání patří třeba *socket()* pro otevření socketu (viz kapitola 5.1), *open()* pro otevření souboru, *write()* pro zápis do souboru a mnohé další.

Kód kernelu můžeme dále rozdělit na ten nezávislý na architektuře a na ten závislý. Nezávislý kód může být i při použití na různých architekturách stejný napříč těmito architekturami; závislý kód se liší.

3.1 Subsystémy

Kernel můžeme dále rozdělit do několika subsystémů, viz obrázek 4.



Obrázek 4: Subsystémy v kernelu

3.1.1 Process Management

Slouží ke správě procesů – v kernelu se jim říká vlákna –, k čemuž můžeme použít systémová volání (*fork()*, *kill()*, *exit()* a další). Součástí je i sdílení zdrojů (CPU) mezi jednotlivými vlákny.

3.1.2 Memory Management

Správa paměti, zde patří např. stránkování nebo virtualizace paměti.

3.1.3 Virtual File System

Zajišťuje použitelnost různých souborových systémů po připojení disků s těmito systémy souborů. Můžeme tak např. pracovat s NTFS oddílem připojeným do OS používajícího Ext4, aniž bychom poznali rozdíl. [28]

3.1.4 Network Stack

Síťový subsystém, který se stará o vše, co souvisí s fungováním daného počítače v síti. Spravuje síťová rozhraní a jejich nastavení. Konfiguraci této části kernelu se věnuje tato práce.

3.1.5 Device Drivers

Ovladače zařízení zajišťují to, že připojený hardware bude fungovat, at'už jde o Bluetooth nebo sériový port.

3.1.6 Arch

Kód závislý na architektuře.

4 Výměna zpráv mezi programy a kernelem

V Linuxu existuje několik rozhraní pro výměnu zpráv (informací) mezi kernelem a user-space, což je nezbytnou podmínkou fungování celého systému. Tato rozhraní umožňují aplikacím číst informace poslané kernelem a kernelu umožňují zpracovávat požadavky od aplikací.

Následující informace jsou čerpány z knihy Understanding Linux Network Internals. [32]

(Poznámka: Rozdělení na „Získání informací“ a „Zápis informací“ není tak striktní, např. procfs umožňuje změny informací v kernelu, Netlink zase nemusí sloužit jen ke změnám konfigurace, ale lze jej použít i k získání současné konfigurace.)

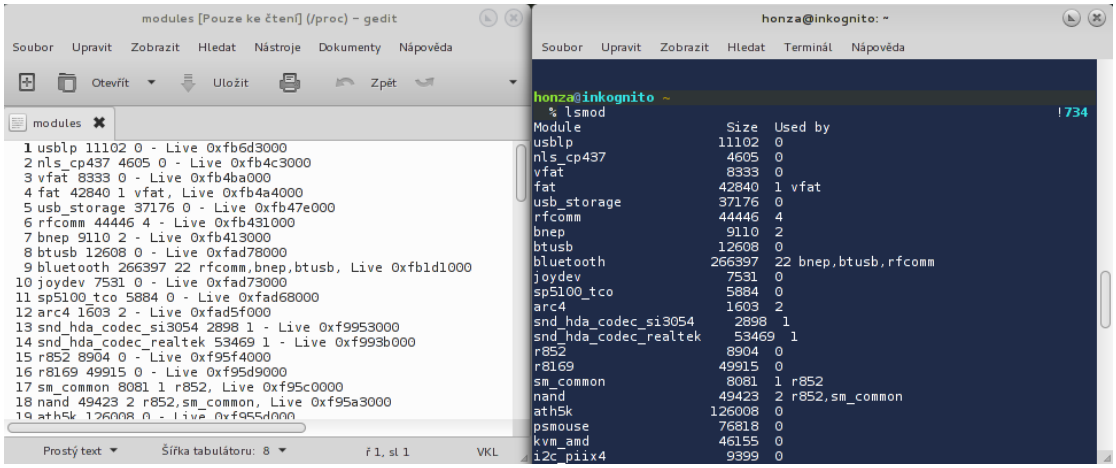
4.1 Získání informací z kernelu

Programátor nebo uživatel může získat požadované informace z kernelu například pomocí jednoho z těchto tří rozhraní:

- procfs
  - jedná se o virtuální souborový systém (viz kapitola 2.3), umožňující kernelu exportovat informace do user-space ve formě souborů; obvykle jsou tyto informace v GNU/Linux k dispozici v adresáři /proc. Ve skutečnosti se nejedná o soubory uložené na disku, ale mohou být čteny a lze do nich i zapisovat. Navíc jim mohou být přiřazena práva jako obvyklým souborům, čímž část kernelu, která tyto soubory vytvořila, může povolit nebo zakázat čtení i zápis různým skupinám uživatelů. V /proc ani v žádném z jeho podadresářů nelze vytvářet soubory ani je mazat, lze jen upravovat soubory, které tam už jsou vytvořeny. Samozřejmě jen pokud k tomu má dotyčný uživatel oprávnění;
  - mnoho nástrojů zobrazujících informace o právě běžícím systému ve skutečnosti nedělá nic jiného, než že čtou informace z adresáře /proc, [35] viz obrázek 5 porovnávající informace ze souboru /proc/modules s výpisem příkazu lsmod,
  - každý právě běžící proces má v adresáři /proc složku pojmenovanou dle svého PID; status daného procesu lze tedy zjistit např. příkazem *cat /proc/PID/status*, [36]
  - tento formát výměny dat je pro změny konfigurace nepraktický, protože vyžaduje parsování dat do dané formy souboru; [46]
- sysctl
  - toto rozhraní umožňuje číst a měnit hodnoty proměnných kernelu, přičemž viditelné jsou zde pouze proměnné, které určil sám kernel. Z user-space je možno k těmto proměnným přistoupit dvěma způsoby – jedním je systémové volání sysctl a druhým je již zmíněný procfs. Kernel, který procfs podporuje, vytvoří speciální složku /proc/sys, obsahující soubor pro každou proměnnou exportovanou kernelem;

  - zatímco procfs exportuje primárně informace sloužící jen pro čtení, většina informací exportovaná sysctl je přepisovatelná. Výběr použití jednoho z těchto dvou rozhraní záleží na tom, jak velké je množství exportovaných informací. Jednoduché datové struktury nebo proměnné jsou exportovány sysctl, zatímco složitější data exportuje procfs (zde patří např. cache nebo statistiky);
  - např. soubory týkající se nastavení IPv4 jsou umístěny v adresáři /proc/sys/net/ipv4;

- sysfs
  - je to modernější nástupce procfs a sysctl,
  - jako u procfs se jedná o virtuální souborový systém. Exportuje množství informací a to velmi čistým a organizovaným způsobem (do složky /sys); jedná se především o informace nesouvisející s procesy: konfigurace kernelu a zařízení;
  - stejně jako u procfs je omezen strukturou souborů a při přenosu dat pak velikostí stránky paměti. [46]



Obrázek 5: soubor modules v /proc vs. lsmod

4.2 Zápis informací do kernelu

Pokud chce programátor změnit konfiguraci určité části kernelu (což je i cíl této práce), může využít následujících rozhraní:

- ioctl
  - je zkratkou pro input/output control a používá se pro konfiguraci různých zařízení, resp. souborů reprezentujících tato zařízení (viz kapitola 2.1),
  - pomocí ioctl konfigurují síťový subsystém kernelu např. příkazy ifconfig nebo route;
- Netlink socket
  - nový mechanismus pro komunikaci se síťovým subsystémem kernelu, blíže si jej popíšeme v následující kapitole,
  - pomocí Netlink socket konfiguruje síťový subsystém např. iproute.

5 Netlink socket

Netlink socket je rozšířením standardního BSD socket API (otevření socketu, odeslání dat, zavření socketu), kdy vytvoříme socket, jako bychom posílali data po síti, ale jako destinaci určíme kernel. (Netlink umožňuje i komunikaci mezi dvěma procesy v user-space nebo je možné použít jej jako komunikační prostředek uvnitř kernelu, tyto vlastnosti ale nejsou předmětem této práce.)

Posílaná data se skládají z Netlink hlavičky doplněné dalšími strukturami, podle toho, čeho konkrétně chceme dosáhnout. Oba konce spojení jsou v Netlink identifikovány buď PID procesu, který otevřel socket (pro user-space) nebo jako 0 (pro kernel).

Netlink dále umožňuje posílat kromě unicastových zpráv i multicastové zprávy, kdy příjemcem zprávy nemusí být jen PID, ale může se jednat i o ID multicastové skupiny nebo o kombinaci obou. Multicastové skupiny se pak používají pro odesílání informací z kernelu, kdy je možno se k nim z user-space přihlásit a přijímat tak informace například o změnách ve směrovacích tabulkách. [43] [32]

Netlink je (na rozdíl třeba od systémových volání nebo ioctl) plně duplexní, což v praxi znamená, že iniciátorem spojení nemusí být jen aplikace v user-space, ale i kernel. [33]

Při psaní programů využívajících Netlink socket je možno využít knihovnu libnl, která poskytuje API pro Netlink a přebírá od programátora úkoly týkající se vytváření socketu, sestavování zpráv a odesílání i přijímání dat. Tato knihovna je poměrně dobře dokumentovaná. [34] V této práci jsem knihovnu nevyužil.

Publikace [46] poskytuje srovnání Netlinku s některými dalšími rozhraními mezi user-space a kernelem; výsledkem je tabulka 1, kde metodikou byly tyto parametry:

- Přenositelnost mezi architekturami
  - některé architektury umožňují použití jiné délky slova (nejmenší počet bitů, které je počítač schopen zpracovat) v user-space a jiné v kernelu. Např. x86_64 umožňuje aplikacím použít v user-space 32bitové slovo, zatímco kernel pracuje s 64bitovým slovem, což může být problém např. při přenosu long integeru, který má pro 32b slovo velikost 4 B, zatímco pro 64b slovo má velikost 8 B. Dá se říci, že všechny typy rozhraní toto zvládají, ale často ne samy od sebe; v takových případech pak rozhraní vyžadují speciální vrstvu, která data převede do odpovídající velikosti;
- Řízení událostmi
  - kernel je při nějaké změně schopen sám tuto změnu propagovat do user-space, takže procesy v user-space nejsou odkázány na pravidelné čtení dat, aby jim žádná změna neunikla;

Rozhraní	Přenositelnost mezi architekturami	Řízení událostmi	Rozšiřitelnost	Přenos velkých dat
system calls	ne	ne	ne	ano
/proc	ano	ne	ne	ne
sysfs	ano	ano	ne	ne
Netlink	ano	ano	ano	ano

Tabulka 1: Srovnání některých rozhraní s Netlinkem

- Rozšiřitelnost
  - pokud nově přidaná vlastnost vyžaduje změnu informace, která se doteď předávala mezi kernelem a user-space, dochází k porušení zpětné kompatibility. Řešením může být přidání redundantních operací, což není ideální stav;
  - informací je myšlena např. pevně daná sekvence hodnot nebo datová struktura;
- Přenos velkých dat
  - možnost přenosu velkého množství dat je vhodá obzlášťpři nutnosti obnovení složité konfigurace.

Popis vytváření socketu pro odeslání dat a popis samotného sestavení Netlink zprávy rozdělím do dvou podkapitol.

5.1 Socket

Sockety jsou jednou z metod meziprocesní komunikace, přičemž procesy nemusí být na stejném počítači; sockety se tedy využívají i při programování síťových aplikací.

Socket je de facto označení pro konec komunikačního kanálu mezi dvěma procesy. [37]

Pokud není uvedeno jinak, zdrojem informací jsou kniha GNU/Linux Application Programming a manuálové stránky kernelu. [38] [39]

API pro jazyk C poskytuje sadu funkcí pro vývoj aplikací typu klient-server.

5.1.1 Vytváření socketů

Prototyp funkce socket() je následující:

```
int socket(int domain, int type, int protocol);
```

Výpis 1: Prototyp funkce socket()

kde:

- *domain* definuje rodinu protokolů, která bude použita ke komunikaci. Samy rodiny jsou definovány v souboru /usr/include/bits/socket.h (adresář /usr/include/ obsahuje hlavičkové soubory pro kompilátor; další v tomto textu zmiňované hlavičkové soubory se nachází ve stejném adresáři, proto tento „prefix“ budu vynechávat) a patří mezi ně například:
  - AF_LOCAL a AF_UNIX pro lokální komunikaci (např. pipes),
  - AF_INET pro IPv4,
  - AF_INET6 pro IPv6,
  - AF_NETLINK pro komunikaci user-space s kernelem;
- *type* určuje styl, kterým bude komunikace probíhat. Mezi možnosti patří například:
  - SOCK_STREAM pro TCP spojení, tedy navázání komunikace před posíláním zpráv, zajištění toho, že se žádná data neztratí a dojdou v pořadí, ve kterém byla odeslána,
  - SOCK_DGRAM pro UDP spojení, kdy výhodou oproti TCP je rychlost, protože příjem dat nemusí být potvrzen a spojení nemusí být navázáno. Nevýhodou je, že není nijak zajištěno, zda data dorazí na místo určení,
  - SOCK_RAW pro odeslání surových dat;
- *protocol* určuje, který konkrétní protokol se použije. Tato volba je výrazně omezena použitým domain a type.

Pro použití Netlink socket je tedy nutné nastavit *domain* jako AF_NETLINK. (Nebo PF_NETLINK; je to v podstatě to samé, protože PF_NETLINK má definovánu hodnotu a AF_NETLINK je následně nadefinováno jako PF_NETLINK. AF je zkratkou pro address family, PF zase pro protocol family. Jedná se o přežitek.) [40]

type může být nastaven jako SOCK_DGRAM nebo SOCK_RAW; dle mých praktických zkušeností v tom není žádný rozdíl.

Nastavení AF_NETLINK jako *domain* omezuje volby *protocol* na následující: [41]

- NETLINK_ROUTE
  - získávání informací o rozhraních (interface), adresách na nich, směrovacích informací apod.,
  - úpravy směrovacích tabulek, změny nastavení rozhraní, IP adres apod.,
  - vytváření rozhraní, přiřazení dalších IP adres na rozhraní, přidávání cest do směrovací tabulky a další,
  - bude dále upřesněno;

- NETLINK_FIREWALL
  - slouží k vytváření a debugování modulů pro iptables (nástroj pro práci se síťovou komunikací) v user-space; [42] [43]
- NETLINK_GENERIC [44]
  - vznikl kvůli obavě o vyčerpání hodnot pro Netlink protokoly,
  - chová se jako Netlink multiplexer;
- NETLINK_W1, NETLINK_USERSOCK, NETLINK_INET_DIAG, NETLINK_NFLOG, NETLINK_XFRM, NETLINK_SELINUX, NETLINK_ISCSI, NETLINK_AUDIT, NETLINK_FIB_LOOKUP, NETLINK_CONNECTOR, NETLINK_NETFILTER, NETLINK_IP6_FW, NETLINK_DNRTMSG a NETLINK_KOBJECT_UEVENT, jejichž účel je k nalezení v manuálových stránkách. [41]

Pro modifikaci některých síťových nastavení kernelu tedy vytvoření socketu bude vypadat (SOCK_DGRAM možno nahradit za SOCK_RAW):

```
int socket(AF_NETLINK, SOCK_DGRAM, NETLINK_ROUTE );
```

Výpis 2: Funkce socket() pro modifikaci síťového nastavení

Tato funkce vrací integer nazývaný file descriptor, což je reference na otevřený soubor, díky čemuž můžeme k tomuto souboru přistupovat. Stejně tak funguje např. funkce open(). V kapitole 2.1 jsem psal o tom, že v UNIXu (a unixových systémech) je vše soubor, tedy i vytvářené spojení je soubor. [45]

Dle manuálových stránek ([41]) nebo některých dalších zdrojů ([43], [60], [61]) je třeba při odesílání použít funkci sendmsg() a pro přijímání funkci recvmsg(), což vyžaduje použití struktur msghdr, sockaddr_nl a iovec. Dle mých zkušeností fungují i pro RTM_GET* požadavky (viz kapitola 5.2) – kdy se dá rozparsováním přijatých dat ověřit, že vše proběhlo v pořádku – funkce send() a recv(), ale použil jsem ty doporučené.

Použité struktury jsou tedy tyto:

```
struct sockaddr_nl {
    __kernel_sa_family_t   nl_family ;
    unsigned short nl_pad;
    __u32      nl_pid ;
     __u32        nl_groups;
};
```

Výpis 3: struktura sockaddr_nl (/linux/netlink.h)

Struktura sockaddr_nl (výpis 3) slouží k popisu Netlink klienta. [41] V praxi jí potřebujeme při odesílání dat do kernelu; v tom případě nastavíme nl_pid na 0 (to znamená, že zprávu zpracuje kernel nebo že jde o multicast). Nebo nenastavíme, 0 je defaultní hodnota. nl_family by vždy měla být nastavena na AF_NETLINK. [43]

```
struct msghdr {
    void *msg_name;
    socklen_t msg_namelen;
    struct iovec *msg_iov;
    size_t msg_iovlen;
    void *msg_control;
    size_t msg_controllen;
    int msg_flags;
};
```

Výpis 4: struktura msghdr (/bits/socket.h)

Struktura msghdr (výpis 4) popisuje zprávu, která bude odeslána. msg_name odkazuje na cílovou adresu, v mém případě kernel, jenž je pro tyto účely popsán strukturou sockaddr_nl. msg_namelen obsahuje velikost cílové adresy, tedy velikost strukury sockaddr_nl. msg_iov odkazuje na strukturu iovec, jež ponese odesílanou Netlink zprávu. msg_iovlen pak obsahuje počet těchto iovec struktur. Ostatní proměnné se nenastavují. [61]

```
struct iovec {
    void *iov_base;
    size_t iov_len;
};
```

Výpis 5: struktura iovec (/bits/uio.h)

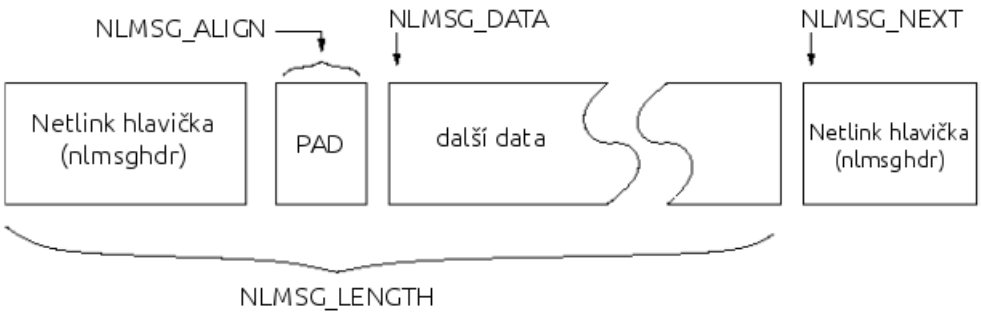
Struktura iovec (výpis 5) popiuje data, která budou odeslána. iov_base odkazuje na Netlink hlavičku a iov_len obsahuje délku zprávy, tedy nlmsg_len, viz kapitola 5.2.

Použití těchto struktur při odesílání ukáži v kapitole 7.

5.2 Netlink

Každá Netlink zpráva začíná Netlink hlavičkou, za níž mohou nebo nemusí následovat další struktury. (Tyto struktury popíši v kapitole 5.3.) Pokud chceme data z kernelu jen číst, stačí nám odeslat jen Netlink hlavičku s patřičně nastavenými proměnnými.

Odesílaná zpráva pak může vypadat třeba jako na obrázku 6 (zdroj původního obrázku: [43]), kde jde vidět, že Netlink zpráv může být i několik najednou.



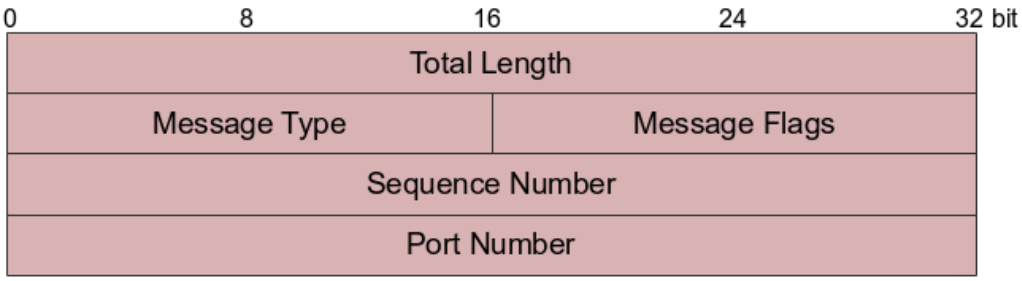
Obrázek 6: Odesílaná Netlink zpráva

Netlink hlavička je struktura nlmsghdr definovaná v /linux/netlink.h, viz výpis 6.

```
struct nlmsghdr {
    __u32    nlmsg_len;
    __u16    nlmsg_type;
    __u16    nlmsg_flags;
    __u32    nlmsg_seq;
    __u32    nlmsg_pid;
};
```

Výpis 6: Netlink hlavička - nlmsghdr

Její přehlednější grafické zobrazení se nachází na obrázku 7 (zdroj: [48]).



Obrázek 7: nlmsghdr

Tato hlavička se tedy skládá z následujících částí: [41] [43] [46]

- nlmsg_len
  - celková délka odesílaných dat včetně této hlavičky. Nejmenší přípustná hodnota je tedy 4 B (sizeof(struct nlmsghdr)), obvykle vyšší (v případě přidružení dalších dat za hlavičkou). Pokud bude tato velikost nastavena špatně, přidružená data mohou být ignorována, i když budou nastavena správně;
- nlmsg_type
  - zde existují dva druhy. Prvním jsou obecné typy zpráv, společné pro všechny dostupné typy Netlink *protocolu* a jsou to:
    - NLMSG_NOOP - tato zpráva je ignorována,
    - NLMSG_ERROR - signalizace chyby, za hlavičkou nlmsghdr se pak nachází struktura nlmsgerr. Tento typ zprávy obvykle odešle kernel do user-space, pokud nastala chyba při zpracování požadavku a uživatel si od kernelu vyžádal ACK;
    - NLMSG_DONE - pokud se v datovém toku nachází více Netlink zpráv najednou, tak zpráva označená takto je poslední,
    - NLMSG_OVERRUN - v současné době se nepoužívá;
  - pak existují typy zpráv závislé na použitém *protocolu*, pro NETLINK_ROUTE to jsou (blíže budou představeny v kapitole 5.3) [49]
    - RTM_NEW* - vytvoření nastavení, které je pro kernel nové (např. přidání druhé IP adresy na rozhraní), nebo změna existujícího nastavení (např. změna existující IP adresy na rozhraní),
    - RTM_GET* - získání informací z kernelu,
    - RTM_DEL* - smazání existujícího nastavení;
- nlmsg_flags
  - podobný případ jako nlmsg_type; mezi společné flagy patří tyto:
    - NLM_F_REQUEST - pokud je zpráva požadavkem (request), musí mít tento flag. Všechny zprávy směřující z user-space do kernelu jsou požadavky, takže všchny musí mít tento flag nastaven, jinak do user-space přijde z kernelu chyba *invalid argument*;
    - NLM_F_MULTI - flag označující data skládající se z více Netlink zpráv za sebou; poslední zpráva má nastaven nlmsg_type na NLMSG_DONE,
    - NLM_F_ACK - vyžádá si potvrzení od kernelu, že požadavek byl úspěšně splněn. Pro svázání požadavku s potvrzením je vhodné nastavit v požadavku sekvenční číslo nlmsg_seq a PID nlmsg_pid;
    - NLM_F_ECHO - přes stejný socket odešle zprávu zpět;

  - pokud je nlmsg_type nastaven na RTM_GET*, pak jsou k dispozici tyto flagy:
    * NLM_F_ROOT - chceme z kernelu získat kompletní tabulky konkrétních dat, ne jen jednu položku. Pokud chceme např. získat informace o rozhraních, kernel na náš požadavek vrátí sekvenci Netlink zpráv (s nastavenými NLM_F_MULTI a NLMSG_DONE, viz výše), přičemž každá nlmsghdr hlavička v odpovědi bude následována informacemi o jednom rozhraní. Počet rozhraní se tedy rovná počtu přijatých Netlink odpovědí. Pokud by ovšem flag NLM_F_ROOT nastaven nebyl, kernel vrátí pouze první položku, na kterou narazí;
    * NLM_F_DUMP - kernel odpoví zprávou obsahující všechny dostupné informace o daném subsystému;
  - pokud je nlmsg_type nastaven na RTM_NEW*, pak jsou k dispozici tyto flagy:
    * NLM_F_REPLACE - pokud nově přidávaná konfigurace, resp. nově přidávaný objekt (např. rozhraní určitého jména či indexu) již v kernelu existuje, pak se při použití tohoto flagu přepíše,
    * NLM_F_EXCL - pokud nově přidávaný objekt už existuje, pak se při použití tohoto flagu nepřepíše,
    * NLM_F_CREATE - pokud přidáváme nový objekt/konfiguraci. Tento flag se používá v kombinaci s oběma flagy výše,
    * NLM_F_APPEND - nová data se připojí za stará data. Vhodné např. při přidávání záznamů do směrovací tabulky;

- nlmsg_seq
  - vhodné při použití flagu NLM_F_ACK pro unikátní identifikaci požadavku. Odpověď (ACK z kernelu) má pak stejné sekvenční číslo jako odeslaný požadavek. Jelikož je Netlink nespolehlivý protokol (viz kapitola 5.1), je někdy vhodné si takto ověřit, že byl daný požadavek obsloužen a nedošlo k jeho ztrátě např. kvůli nedostatku paměti;
  - ACK z kernelu přijde ve formě struktury nlmsghdr následované strukturou nlmsgerr, která v případě úspěchu bude mít nastaveno pole *error* na 0;

- nlmsg_pid
  - označuje unicastovou adresu Netlink socketu; možno nastavit ve struktuře sockaddr_nl (viz kapitola 5.1),
  - jde o označení původce zprávy.

Na obrázku se znázorněním Netlink zprávy (obrázek 6) je možno vidět i zatím nevysvětlené ukazatele NLMSG_*. Jedná se o makra definovaná ve stejném hlavičkovém souboru jako Netlink hlavička a slouží k vytváření zpráv nebo jejich parsování v případě, že kernel odpoví na požadavek např. konfigurací, kterou odešle pod nlmsghdr hlavičkou.

Makra, která ulehčují přístup k datům v Netlink zprávě a práci s nimi, jsou tato: [43] [50]

- NLMSG_ALIGN(len)
  - zaokrouhlí délku zprávy (parametr len) nahoru na nejbližší hranici NLMSG_ALIGNTO a vrátí novou, zaokrouhlenou délku dat,
  - jedná se o zaokrouhlení po 4 B,
  - toto makro se nepoužívá přímo, ale využívají jej ostatní makra;
- NLMSG_LENGTH(len)
  - pomocí tohoto makra se nastavuje proměnná nlmsg_len v hlavičce nlmsghdr, která určuje délku odesílané zprávy,
  - parametr len zde představuje délku dat připojených za Netlink hlavičkou; makro k této délce přidá délku samotné hlavičky a zaokrouhlí ji;
- NLMSG_SPACE(len)
  - vrací zaokrouhlenou délku dat v bajtech, kterou bude zabírat zpráva o délce dat len,
  - odpovídá NLMSG_ALIGN(NLMSG_LENGTH(len));
- NLMSG_DATA(nlh)
  - parametrem makra je hlavička nlmsghdr (zde parametr nlh), makro pak vrací ukazatel na data za touto hlavičkou;
- NLMSG_NEXT(nlh,len)
  - používá se, když potřebujeme rozparsovat data přijatá od kernelu, např. informace o IP adresách na rozhraních, která se skládají z více Netlink zpráv najednou,
  - parametry makra jsou aktuální nlmsghdr hlavička a délka dat za touto hlavičkou,
  - makro vrací ukazatel na další Netlink hlavičku v pořadí, je tedy dobré použít cyklus, abychom přečetli všechny hlavičky;
- NLMSG_OK(nlh,len)
  - vrací *true*, pokud hlavička i data za ní přišly v pořádku a je možné s nimi dále pracovat (rozparsovat je);
- NLMSG_PAYLOAD(nlh,len)
  - vrací délku dat za Netlink hlavičkou.

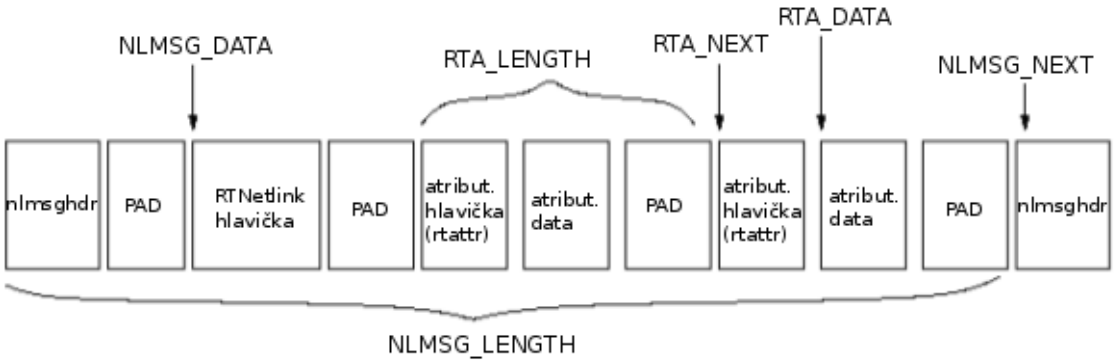
5.3 RTNetlink

V sekci 5.1 jsem ukázal vytváření socketu, kde je pro změnu síťové konfigurace třeba použít NETLINK_ROUTE jako parametr *protocol*. Toto rozšíření standardní Netlink zprávy se označuje jako RTNetlink a podle toho, k čemu má sloužit, jej můžeme rozdělit do skupin: [49]

- manipulace se síťovými rozhraními,
- manipulace s adresami na síťových rozhraních,
- manipulace se směrovací tabulkou,
- manipulace s ARP tabulkou,
- manipulace se směrovacími pravidly,
- manipulace s datovým tokem.

Každá tato skupina má svou vlastní RTNetlink hlavičku, která je při odesílání zprávy do kernelu – a ostatně i v odpovědi kernelu na náš RTM_GET* požadavek – umístěna ihned za Netlink hlavičkou. Navíc je v případě potřeby možné (a často nutné) za tuto RTNetlink hlavičku přidat ještě další strukturu dat, přidružené atributy. Tyto atributy pak obsahují data v závislosti na dané RTNetlink hlavičce a jsou de facto nosičem informací, které chceme v kernelu změnit nebo do něj přidat.

Stejně jako u Netlink zprávy i zde ukáži grafickou reprezentaci dat posílaných do kernelu (obrázek 8, zdroj původního obrázku: [43]).



Obrázek 8: Odesílaná RTNetlink zpráva

RTNetlink hlavičky jsou struktury definované (většinou) v /linux/rtnetlink.h. Jak už jsem psal, hlaviček je více druhů, takže v následujících podsekcích popíši ty, se kterými jsem pracoval.

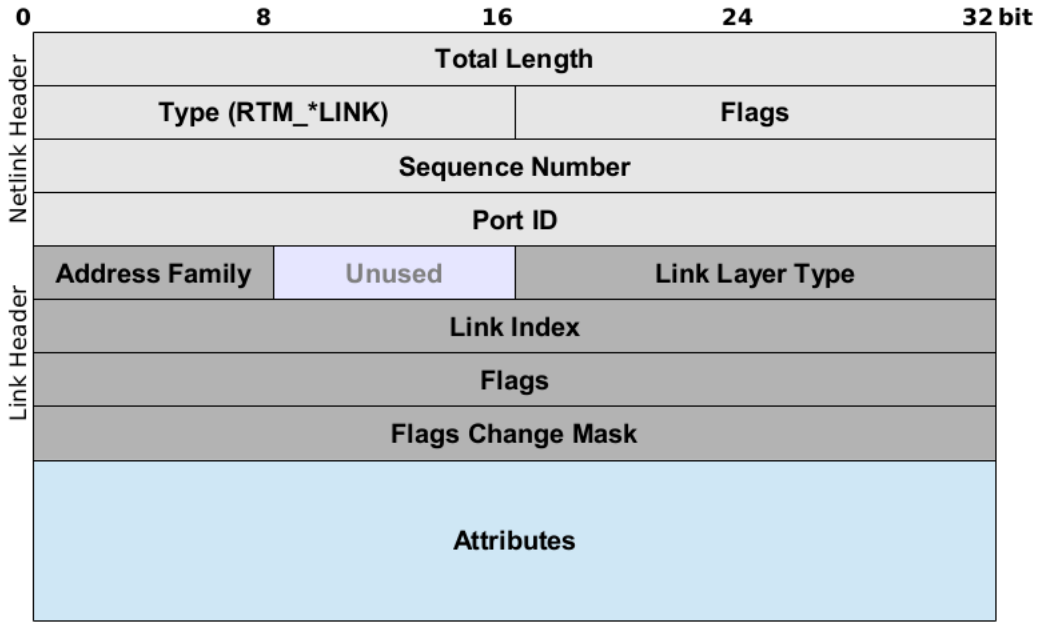
5.3.1 Manipulace se síťovými rozhraními

Pro vytváření, modifikaci a mazání (virtuálních) síťových rozhraní či získání informací o nich existuje struktura ifinfomsg, definovaná následovně (výpis 7):

```
struct ifinfomsg {
    unsigned char  ifi_family ;
    unsigned char  __ifi_pad ;
    unsigned short ifi_type;
    int            ifi_index ;
    unsigned       ifi_flags ;
    unsigned       ifi_change ;
};
```

Výpis 7: RTNetlink hlavička - ifinfomsg

Přehledné grafické znázornění ukazuji na obrázku 9 (zdroj původního obrázku: [51]). Jako první vidíme Netlink hlavičku jako na obrázku 7 a hned za ní RTNetlink hlavičku. V předchozí kapitole jsem psal, že použití NETLINK_ROUTE omezuje nastavení typu zpráv v Netlink hlavičce. To samé dělá i vybraná RTNetlink hlavička. Jak je tedy patrné z obrázku 9, v tomto případě jsme v proměnné nlmsg_type omezeni na tři typy zpráv – RTM_NEWLINK, RTM_DELLINK a RTM_GETLINK. Někde je možné se setkat i s typem RTM_SETLINK, ale jeho funkci lze nahradit typem RTM_NEWLINK s nastaveným nlmsg_flags na NLM_F_REPLACE.



Obrázek 9: ifinfomsg

Za strukturou ifinfomsg ještě následují atributy, těm se ale budu věnovat později.

ifinfomsg se tedy skládá z těchto částí: [49] [51]

- ifi_family
  - definuje rodinu IP adres, obvykle nastaveno jako AF_UNSPEC,
  - pokud získáváme informace z kernelu, můžeme si takto omezit rohraní, o kterých nám kernel pošle informace – AF_INET pro IPv4 nebo AF_INET6 pro IPv6;
- ifi_type
  - používá se jen u zpráv z kernelu do user-space a označuje typ daného síťového rozhraní,
  - hodnota odpovídá konstantám ze souboru /linux/if_arp.h, např. ARPHRD_ETHER pro Ethernet;
- ifi_index
  - jednoznačný identifikátor rozhraní,
  - můžeme díky němu získat z kernelu informace o jednom rozhraní namísto o všech; dále je nutné jej nastavit, pokud chceme změnit konfiguraci na konkrétním rozhraní,
  - v nových verzích Linuxu (od verze 3.7) je takto možné určit index nově vytvářeného rozhraní; jinak jej kernel určí sám;
- ifi_flags
  - v informacích z kernelu je v tomto poli uložen aktuální stav daného rozhraní. Ve zprávách odesílaných do kernelu zde nastavujeme flagy pro nové rozhraní nebo upravujeme flagy na stávajícím rozhraní;
  - dostupné flagy jsou stejné jako pro ioctl rozhraní, jejich seznam je umístěn v manuálových stránkách netdevice [52] a patří tam:
    - IFF_UP - rozhraní je „nahozeno“, tedy aktivní; může přijímat i odesílat pakety,
    - IFF_BROADCAST - rozhraní je schopno poslat pakety na všechny adresy dané podsítě,
    - IFF_MULTICAST - doplňující flag, označující, že rozhraní je schopno multicastu (broadcast je jeho speciální formou). Pokud ale tento flag chybí, tak to neznamená, že by rozhraní nebylo schopno zpracovat multicastový provoz; [53]

- ifi_change
  - dle manuálových stránek kernelu je toto pole určeno pro použití v budoucnu a nemá se měnit, resp. má být vždy nastaveno na 0xffffffff,
  - dle dokumentace libnl má toto pole speciální význam při notifikacích o změnách na rozhraní,
  - při odesílání nové konfigurace do kernelu se toto pole chová jako maska těch flagů, které mají být změněny; jinak řečeno, může být změněn jenom ten flag, na jehož pozici v této masce je jednička (protože flagy jsou vlastně jen hodnoty, viz jejich definice v souboru /linux/if.h), ostatní jsou ignorovány,
  - proto nastavení na hodnotu 0xffffffff bude fungovat vždy; druhou a poslední možností je nastavit ifi_change stejně jako ifi_flags.

5.3.2 Manipulace s adresami na síťových rozhraních

Pro modifikaci, vytváření či mazání IP adres na rozhraních – ať už virtuálních či fyzických – nebo získání informací o nich slouží struktura ifaddrmsg, která je definovaná v souboru /linux/if_addr.h a to následovně (výpis 8):

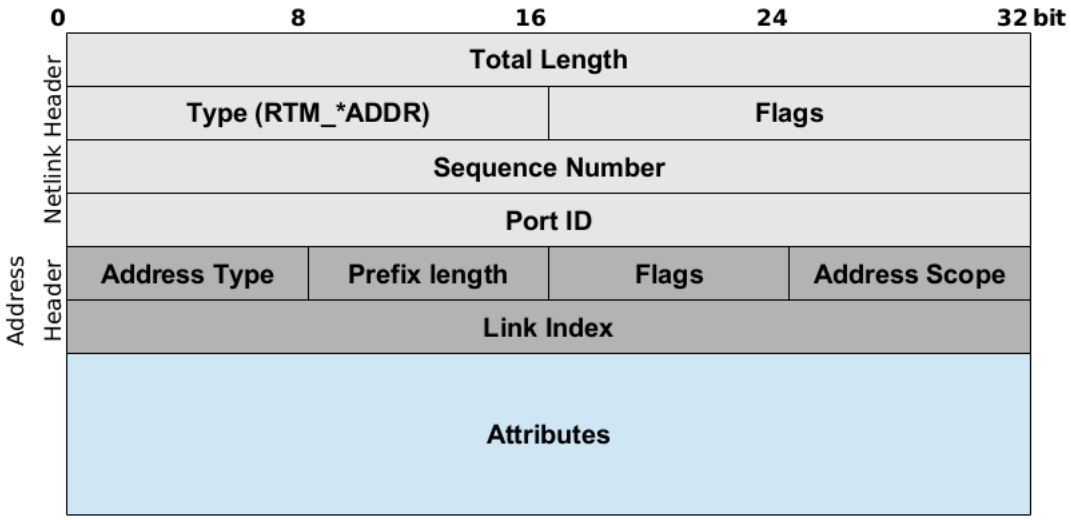
```
struct ifaddrmsg {
    __u8        ifa_family ;
    __u8        ifa_prefixlen ;
    __u8        ifa_flags ;
    __u8        ifa_scope;
    __u32       ifa_index ;
};
```

Výpis 8: RTNetlink hlavička - ifaddrmsg

S grafickým znázorněním na obrázku 10.
V proměnné nlmsg_type v Netlink hlavičce jsme stejně jako v minulém případě omezeni na tři typy, tentokrát na RTM_NEWADDR, RTM_DELADDR a RTM_GETADDR.

Částmi ifaddrmsg jsou: [49] [43] [53]

- ifa_family
  - stejně jako u struktury ifinfomsg jde o rodinu adres, na výběr jsou dvě – AF_INET pro IPv4 a AF_INET6 pro IPv6,
  - při požadavku RTM_GETADDR je nastavením této proměnné možno určit, zda chceme získat informace o adresách IPv4, nebo jen o IPv6,
  - při změně adresy na rozhraní je nastavení této proměnné nutné, jinak je požadavek ignorován;
- ifa_prefixlen
  - délka prefixu, tedy počet bitů síťové části adresy; defaultní hodnota je 0;



Obrázek 10: ifaddrmsg

- ifa_flags
  - např. flag IFA_F_SECONDARY je nastaven, pokud je adresa vedlejší (sekundární) adresou na rozhraní. Adres na rozhraní může více, ale jako sekundární adresa jsou označeny jen ty adresy, které jsou ze stejné sítě jako již nastavená (primární) adresa. Sekundární adresa není vybrána jako zdrojová adresa při odesílání paketu. Po smazání primární adresy se smaže automaticky i ta sekundární, opačně to neplatí; [53]
  - mezi další flagy patří třeba IFA_F_PERMANENT, označující, že daná IPv6 adresa vznikla konfigurací uživatele a ne bezstavovou konfigurací (automatickým nastavením IPv6 adresy na rozhraní pomocí jeho MAC adresy),
  - flagy při vytváření či změně adresy nastavuje kernel, měnit je nelze a jejich případné nastavení v RTNetlink hlavičce bude při odeslání požadavku RTM_NEWADDR ignorováno,
  - ifa_flags tedy můžeme využít jen při čtení odpovědi na požadavek RTM_GETADDR;
- ifa_scope
  - v podstatě jakýsi dosah adresy, oblast, ve které je adresa platná,
  - struktura ifaddrmsg zde přebírá možnosti nastavení rtm_scope ze struktury rtmsg (kapitola 5.3.3),

  - výchozí hodnota (při nenastavení ifa_scope) je *global*, což odpovídá nastavení na RT_SCOPE_UNIVERSE, resp. na hodnotu 0. Jde o adresu platnou kdekoliv, přičemž se nerozlišuje, zda jde či nejde o veřejnou IPv4 adresu. Další možností je RT_SCOPE_HOST, která říká, že adresa je platná pouze zařízení, na kterém je nastavena, a ostatní zařízení v síti k ní přístup nemají; v praxi jde o loopback rozhraní. RT_SCOPE_LINK zase označuje adresu platnou jen na dané LAN, zde patří broadcast adresa. Kernel takto adresu loopbacku a broadcast adresu označí automaticky;
  - využití nachází ifa_scope např. při výběru IP adresy, která se použije jako zdrojová pro odchozí paket, pokud je IP adres nastaveno více. Tato zdrojová adresa se pak vybere podle *scope* cílové adresy, čímž se zajistí, že odpověď bude moct dorazit na stejnou adresu; [64]

- ifa_index
  - identifikuje rozhraní, viz ifi_index v kapitole 5.3.1,
  - při přidávání IP adresy je třeba určit, na které rozhraní chceme adresu přidat,
  - při získávání informací z kernelu můžeme omezit odpověď na jediné rozhraní.

Stejně jako u ifinfomsg mohou za touto hlavičkou následovat atributy.

5.3.3 Manipulace se směrovací tabulkou

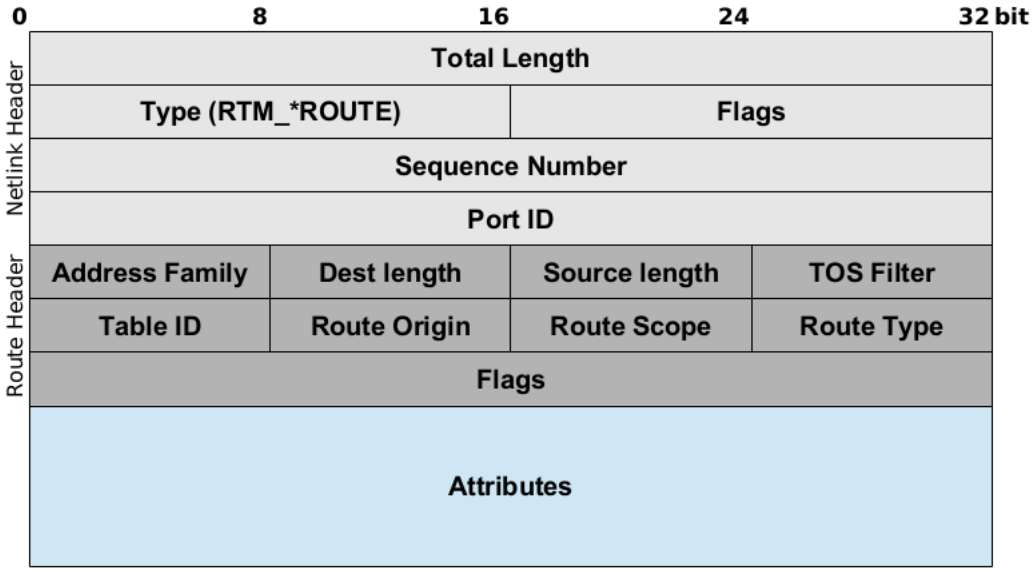
Poslední struktura, se kterou jsem pracoval, je rtmsg a je definována v souboru rtnetlink.h následujícím způsobem (výpis 9):

```
struct rtmsg {
    unsigned char    rtm_family;
    unsigned char    rtm_dst_len;
    unsigned char    rtm_src_len;
    unsigned char    rtm_tos;

    unsigned char    rtm_table;
    unsigned char    rtm_protocol;
    unsigned char    rtm_scope;
    unsigned char    rtm_type;

    unsigned    rtm_flags;
};
```

Výpis 9: RTNetlink hlavička - rtmsg

S grafickým znázorněním na obrázku 11.

Stejně jako v předchozích případech i zde je omezena volba nlmsg_type, tentokrát na RTM_NEWROUTE, RTM_DELROUTE a RTM_GETROUTE.



Obrázek 11: rtmsg

Částmi rtmsg jsou: [43] [49] [53]

- rtm_family
  - rodina adres, stejně jako v předchozích případech (možnosti AF_INET a AF_INET6),
  - pro RTM_NEWROUTE nutné nastavit, jinak je požadavek ignorován; pro RTM_GETROUTE možno použít jako filtr;
- rtm_dst_len
  - délka prefixu adresy cílové sítě, pro RTM_NEWROUTE nutné nastavit;
- rtm_src_len
  - délka prefixu adresy zdrojové sítě; používá se pro nastavení směrovacích pravidel při použití více směrovacích tabulek; není předmětem této práce;
- rtm_tos
  - Type of Service;
- rtm_table
  - do které tabulky se záznam vloží. Defaultní je tabulka Main; další tabulkou je např. Local, kam si kernel ukládá cesty pro lokální a broadcast adresy při přidání adresy na rozhraní;

- rtm_protocol
  - označuje protokol, který danou cestu do kernelu nainstaloval; RTPROT_KERNEL se používá při vzniku cesty autokonfigurací (jako u cesty k výchozí bráně ve výpisu 10), manuálně přidané cesty mají mít *protocol* nastavený jako RTPROT_BOOT,
  - slouží k omezení operací nad cestami podle zdroje jejich vytvoření; [65]
- rtm_scope
  - vzdálenost do cílové sítě,
  - defaultní hodnotou je RT_SCOPE_UNIVERSE a slouží pro cesty ke vzdáleným a nepřímo připojeným cílům. RT_SCOPE_HOST zase označuje cestu, jež vede na stejné zařízení. RT_SCOPE_LINK vede na adresu v lokální síti,
  - slouží také ke kontrole konfigurace. Při odesílání paketu na cílovou adresu, která není dostupná přímo, je třeba odeslat paket na výchozí bránu, která musí být k cíli blíže než odesílatel paketu. Jinak řečeno, *scope* vedoucí k cíli musí být větší než ten vedoucí k výchozí bráně, viz výpis 10, kde je výchozí brána (192.168.1.1) dosažitelná v rámci *scope link*, kdežto cesta do Internetu vede přes defaultní cestu se *scope global*, neboli *universe*. Jelikož jde o defaultní hodnotu pro cesty, tak ve výpisu není obsažena; [64]

```
% ip route sh
default via 192.168.1.1 dev wlan0 proto static
192.168.1.0/24 dev wlan0 proto kernel  scope link  src 192.168.1.101
```

Výpis 10: Výpis směrovací tabulky

- rtm_type
  - obvyklé směrovací tabulky obsahují jen cesty s *type* nastaveným na RTN_UNICAST, což znamená, že jde o přímou či nepřímou (přes výchozí bránu) cestu na nějakou cílovou adresu; [65]
- rtm_flags
  - obvyklé cesty se tímto polem vůbec nezabývají.

5.3.4 Atributy

Za každou z těchto tří RTNetlink hlaviček je třeba vložit atributy. Jako jejich hlavičky je možno použít buď strukturu rtattr (výpis 11, definována v souboru /linux/rtnetlink.h) nebo strukturu nlattr (výpis 12, definována v souboru /linux/netlink.h).

```
struct rtattr {
    unsigned short rta_len;
    unsigned short rta_type;
};
```

Výpis 11: Atributy - rtattr

```
struct nlattr {
    __u16           nla_len;
    __u16           nla_type;
};
```

Výpis 12: Atributy - nlattr

Obě struktury jsou tedy v podstatě stejné, ovšem rtattr má výhodu ve větším počtu definovaných maker (některá makra jdou vidět na obrázku 8). Struktura nlattr některá makra má, některá ne. Nenašel jsem žádný zdroj popisující důvod vzniku této struktury, ale domnívám se, že důvodem bylo zavedení tzv. vnořených (*nested*) atributů (viz kapitola 5.3.4.1), protože struktura nlattr s vnořenými atributy počítá (viz /linux/netlink.h), na rozdíl od strukury rtattr. Z důvodu této relativní „novosti“ jsem se rozhodl použít nlattr, i když se poté ukázalo, že rozdíl nebyl de facto žádný, protože (zřejmě) plánovaná změna zůstala – nejspíše s ohledem na zpětnou kompatibilitu – na půli cesty. K vnořeným atributům se vrátím později.

Částmi nlattr jsou: [49]

- nla_len
  - délka atributu, tedy délka hlavičky + délka připojených dat,
  - délka dat se stejně jako v předchozích hlavičkách zaokrouhluje na 4 B; a ačkoliv má být dle definice nla_len rovno délce dat *bez* zaokrouhlení, v praxi jsem na rozdíl nenarazil a používal jsem zaokrouhlenou délku;
- nla_type
  - typ atributu,
  - typů je velká spousta a lze použít jen ty, které typem operace odpovídají použité RTNetlink hlavičce,
  - jednotlivé typy představím níže.

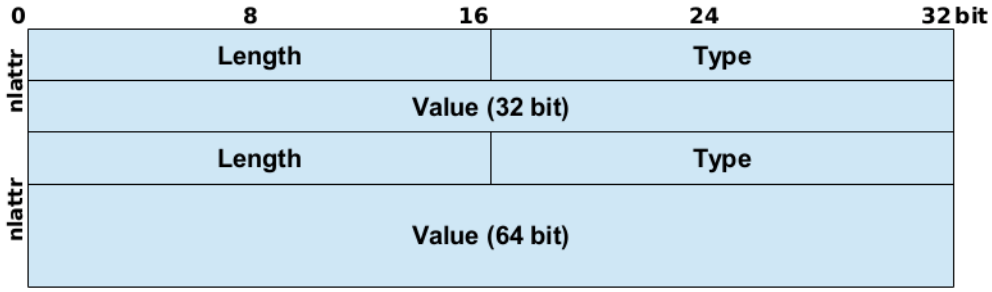
Jelikož má rtattr definováno více maker a obě struktury se chovají stejně, vypíši tedy ty jeho, přičemž platí, že pro použití u strukury nlattr stačí definici makra zkopírovat a zaměnit *rt* za *nl*. Makra jsou tedy následující: [54]

- RTA_OK(rta, attrlen)
  - ověřuje integritu dat za hlavičkou (u každého atributu rta), vrací *true*, pokud jsou data v pořádku,
  - attrlen je délka dat za za danou RTNetlink hlavičkou,
  - pokud nevrací *true*, předpokládáme, že žádná data už nenásledují, i kdyby attrlen byl nenulový,
  - využití při parsování odpovědi na RTM_GET*;
- RTA_DATA(rta)
  - vrací pointer na data za danou rta hlavičkou;
- RTA_PAYLOAD(rta)
  - vrací délku dat za hlavičkou atributu rta;
- RTA_NEXT(rta, attrlen)
  - vrací pointer na další atribut v sekvenci atributů a zároveň upravuje hodnotu attrlen (délky dat za RTNetlink hlavičkou),
  - používá se v kombinaci s RTA_OK při parsování odpovědi kernelu;
- RTA_LENGTH(len)
  - len je délka dat a toto makro vrací délku atributu, tedy hlavičky + data, bez zaokrouhlení,
  - toto makro by se oficiálně mělo používat pro nastavení rta_len (či nla_len), ale jak jsem psal výše, nenarazil jsem na rozdíl ve funkčnosti mezi tímto a makrem RTA_ALIGN resp. NLA_ALIGN, které vrací tuto délku zaokrouhlenou;
- RTA_SPACE(len)
  - vrací skutečnou délku (se zaokrouhlením), kterou bude daný atribut ve zprávě zabírat.

K lepšímu pochopení RTM_GET* požadavku a parsování odpovědi uvedu kroky, které je potřeba udělat. V následujícím textu budu vysvětlovat jednotlivé části programů sloužících k vytvoření tunelu, ale ani jeden z nich s RTM_GET* zprávou nepracuje a parsování by tak zůstalo nevysvětleno. Postup kroků pro (například) získání informací o rozhraních by byl tento (vytvoření socketu vynechávám):

1. Budeme odesílat zprávu bez atributů (nic nenastavujeme), v tomto případě strukturu nlmsghdr následovanou strukturou ifinfomsg. Nastavení proměnných v nlmsghdr je:
   (a) nlmsg_type = RTM_GETLINK; tedy chceme získat informace o rozhraních,
   (b) nlmsg_flags = NLM_F_REQUEST | NLM_F_ROOT; posíláme do kernelu požadavek a chceme získat všechny informace (o rozhraních), které kernel má,
   (c) nlmsg_len = NLMSG_LENGTH(sizeof(struct ifinfomsg)); nastavíme velikost zprávy jako velikost nlmsghdr + velikost ifinfomsg;
2. Nastavení proměnných v ifinfomsg je volitelné. Můžeme určit rodinu adres nebo číslo rozhraní, z něhož chceme získat informace. I když zde nic nenastavíme a pro tento účel tuto strukturu vlastně ani v programu nemusíme vůbec mít, je třeba s ní počítat v poli nlmsg_len, jinak se program neprovede.
3. Zprávu odešleme pomocí funkce sendmsg() a přijmeme odpověď pomocí funkce recv().
4. Přijatá zpráva je zanořením struktur podobná té odesílané, jen navíc obsahuje atributy. Postupujeme od vnější strukury k vnitřním takto:
   (a) Kvůli možnosti, že odpověď obsahuje více Netlink zpráv (viz obrázek 6; ve skutečnosti se počet zpráv rovná počtu rozhraní), vytvoříme cyklus, který prochází jednotlivé hlavičky (pomocí makra NLMSG_NEXT a kontroly délky zbývajících dat (nebo možno použít makro NLMSG_OK);
   (b) Za každou Netlink hlavičkou se bude nacházet RTNetlink hlavička, pointer na ni získame makrem NLMSG_DATA. Z těchto dat už můžeme získat např. index daného rozhraní (přes ifi_index);
   (c) Nyní už se jen zbývá dostat k atributům. V souboru /linux/if_link.h je definováno makro IFLA_PAYLOAD, které vrací pointer na data (na hlavičku prvního atributu) za ifinfomsg hlavičkou. Analogicky funguje makro IFA_PAYLOAD pro strukturu ifaddrmsg (/linux/if_addr.h) atd.;
   (d) Procházíme jednotlivé atributy: můžeme je např. v cyklu zkontrolovat makrem RTA_OK a poté získat pointer na další atribut pomocí RTA_NEXT. Zde už pomocí rta_type získáme typ daného atributu a podle toho získáme hodnotu daného typu;
   (e) Pokud nezbývá žádný další atribut, vrátíme se k bodu (a) a rozparsujeme další Netlink hlavičku. Pokud žádná další hlavička neexistuje, končíme.

Na obrázku 12 jde vidět, jak mohou vypadat atributy v odesílané zprávě. Za hlavičkou (pole Length a Type) se nachází konkrétní data pro daný typ. Délka prvního atributu pak je 64 bitů, tedy 8 bajtů (hlavička + data), délka druhého atributu 96 bitů, tedy 12 bajtů.

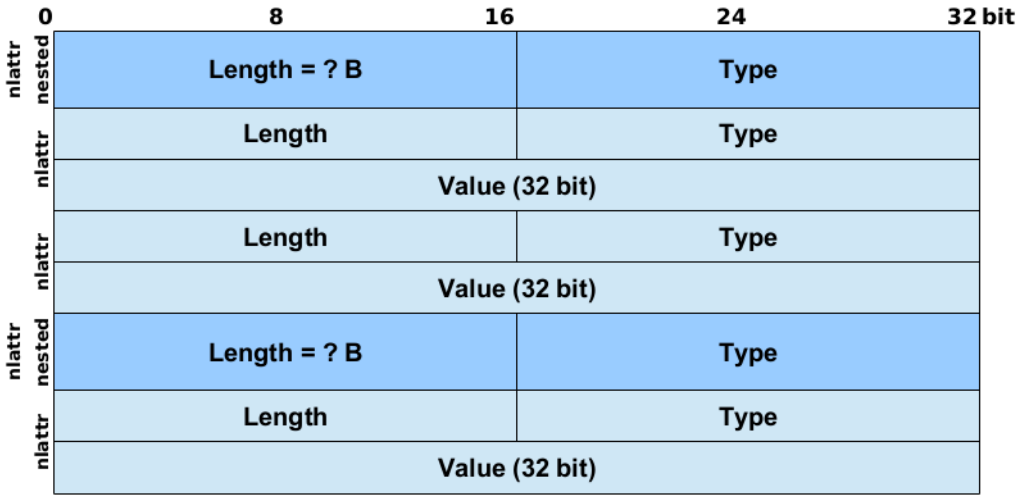


Obrázek 12: Ukázka použití dvou atributů

Tento formát umožňuje jednoduchou rozšiřitelnost. Při zpracování atributů se kontroluje jeden atribut po druhém a následná akce závisí na typu daného atributu, takže po přidání nového atributu v kernelu je třeba jen lehce upravit user-space aplikaci, aby s novým atributem uměla pracovat. Navíc zůstává zpětná kompatibilita, protože pokud aplikace s novým atributem nepracuje, pak je jeho existence ignorována. [46]

Na pořadí atributů nezáleží, postupně se projdou všechny. Nelze ale přesouvat atributy z jednoho vnořeného atributu do druhého.

5.3.4.1 Vnořené atributy Vnořený atribut je atribut, který místo dat za hlavičkou obsahuje další atribut. Více přehledně to asi půjde vidět na obrázku 13.



Obrázek 13: Vnořené atributy

Vnořené atributy slouží k jakémusi zabalení specifických (společných) atributů k sobě. Typickým příkladem by byl vnořený atribut sloužící k informacím o IP adresách. Jeho obsahem by pak byl např. jeden atribut s IPv4 adresou, druhý atribut s IPv6 adresou atd. [46] Vnořené atributy jsou potřeba při vytváření rozhraní, protože jedním vnořeným atributem se určí typ rozhraní a dalším pak dodatečné informace potřebné k vytvoření tohoto rozhraní.

Délka, resp. „dosah“ vnoření atributů je dán hodnotou pole nla_len v hlavičce atributu, který obsahuje další atributy. Proto, pokud neznáme hodnotu nla_len prvního atributu na obrázku 13, tak jen pomocí něj nemůžeme říct, kam až sahá vnoření. Hlavní scénáře jsou dva:

1. nla_len u prvního atributu bude nastavena na 20 B. To by znamenalo, že v odesílané zprávě (pokud by v ní nebyly žádné další atributy, než jaké jsou na obrázku) jsou:
   - vnořený atribut obsahující dva atributy,
   - vnořený atribut obsahující jeden atribut; délka tohoto vnořeného atributu by byla 12 B;
2. nla_len u prvního atributu bude nastavena na 32 B. V tomto případě odesílaná zpráva obsahuje:
   - vnořený atribut obsahující tři atributy, z nich poslední je také vnořený atribut, pro změnu obsahující jeden atribut.

Zde se dostávám k výše zmiňovanému rozdílu mezi rtattr a nlattr. U definice struktury nlattr je v souboru /linux/netlink.h vytvořeno makro NLA_F_NESTED, které by mělo vnořený atribut označit tak, že v poli nla_type nastaví bit s nejvyšší vahou na hodnotu 1. Pokud se toto provede, vnořený atribut se vytvoří, nicméně úplně stejně se vytvoří i bez tohoto makra, resp. nastaveného bitu. Vnořené atributy tak můžeme vytvářet pomocí obou struktur.

5.3.4.1.1 Chybějící dokumentace Jak je celý Netlink socket docela špatně (či spíše nekompletně nebo rovnou zastarale) dokumentován, tak s atributy, tím spíš těmi vnořenými, je to ještě horší. Ve starších (několik let) článcích, které obsahují příklady pro nastavení různých částí síťového subsystému (např. [33]), se obvykle vnořené atributy vůbec nevyužívají, což velmi často zapřičiňuje nefunkčnost těchto příkladů na dnešních verzích kernelu. Nefunkční mohou být dokonce příklady přímo v manuálových stránkách kernelu, např. ten pro nastavení MTU v [54]. Dokonce ani na místě, kde jsou atributy definovány (atributy potřebné pro nastavení rozhraní jsou v souboru /linux/if_link.h), nejde poznat, k čemu atributy slouží, natož které z nich se používají jako vnořené.

Při programování pak v podstatě neexistuje jiná možnost než si najít informace třetích stran, kterých není mnoho (čerpal jsem například z článku řešícího problém vytvoření VLAN rozhraní, viz [55]; odtud jsem aplikoval vytváření vnořených atributů), nebo problém pochopit ze zdrojových kódů. Jeden z autorů Netlink socket, Alexey Kuznetsov [56], je rovněž původním autorem balíku iproute2 [57], jenž slouží k nastavování síťového subsystému kernelu. Iproute2 využívá právě Netlink socket, takže jeho zdrojové kódy mohou sloužit jako návod, které atributy použít pro který konkrétní účel.

5.3.4.2 Typy atributů Atributy se liší dle použité RTNetlink hlavičky. Některé z nich jsou vysvětleny v manuálových stránkách [49], některé vysvětleny nejsou. Následuje seznam některých atributů pro základní představu.

5.3.4.2.1 Atributy pro ifinfomsg (síťová rozhraní) Atributy, které mohou být použity za strukturou ifinfomsg, jsou definovány v souboru /linux/if_link.h, pro GRE tunely pak v /linux/if_tunnel.h.

Patří sem:

- IFLA_IFNAME - jméno rozhraní (např. eth0, wlan0),
- IFLA_MTU - maximální velikost paketu, a další.

5.3.4.2.2 Atributy pro ifaddrmsg (IP adresy na rozhraních) Tyto atributy jsou definovány v souboru /linux/if_addr.h.

Patří sem:

- IFA_LOCAL - lokální adresa rozhraní,
- IFA_BROADCAST - broadcast adresa, a další.

5.3.4.2.3 Atributy pro rtmsg (směrovací tabulky) Tyto atributy jsou definovány v souboru /linux/rtnetlink.h.

Patří sem:

- RTA_OIF - výstupní rozhraní pro danou cestu,
- RTA_DST - cílová síť, a další.

6 Tunely v sítích

Tunely jsou obvykle virtuální dvoubodová spojení, sloužící k přenosu paketů jednoho protokolu uvnitř druhého protokolu; typickým příkladem může být přenos např. IPv6 datagramů zabalených do IPv4 hlavičky kvůli přenosu po páteřní IPv4 síti, jež rozděluje dva IPv6 ostrovy. Další využití nacházejí při vytváření VPN, např. pro vzdálený přístup do školní sítě, tedy pro simulaci privátních sítí přes Internet.

V mém případě je k vytvoření tunelu je potřeba vytvořit virtuální rozhraní na obou zařízeních (mezi kterými chci tunel vytvořit) a určit oba konce tunelu. Jako začátek a konec tunelu obvykle slouží interface, kterým se odesílají pakety „ven“, typicky do Internetu.

Sám tunel se chová, jako by šlo o jediný hop, tedy propoj mezi dvěma směrovači, nezávisle na tom, kolik jich mezi zdrojem a cílem ve skutečnosti je.

6.1 GRE tunely

GRE je zkratkou pro Generic Routing Encapsulation a jedná se o obecný tunelovací mechanismus, který je schopen enkapsulovat velké množství různých protokolů a přenášet je po IP síti. Byl vyvinut společností Cisco. [68]

Pakety jsou zabaleny do GRE hlavičky, která je poté obalena hlavičkou linkového protokolu a odeslána z rozhraní začátku tunelu směrem ke konci tunelu, kde dojde k rozbalení a původní paket dále pokračuje svou cestou.

Při vytváření tunelů (viz další kapitola) můžeme vytvořit dva typy GRE tunelu – první bude enkapsulovat data s hlavičkou třetí vrstvy (tedy IP datagram, jedná se o IPoGRE) a druhý bude enkapsulovat data druhé vrstvy (tedy rámec, jedná se o EoGRE). Přenos ethernetového rámce v GRE slouží ke vzdálenému přemostění sítě.

GRE tunely jsou samy o sobě nechráněny (data v nich nejsou šifrována), ale lze je zkombinovat s IPSec. Data pak budou při přenosu přes cizí síť (Internet) šifrována a tím zabezpečena.

7 Vytváření tunelů

Mým úkolem bylo zjistit, jakým způsobem se pomocí Netlink socket vytvářejí GRE tunely, tento postup aplikovat a zdokumentovat jej.

7.1 IP over GRE

Z výše popsaného teoretického základu je možno úspěšně vytvořit tunel mezi dvěma počítači a tento pak provozovat.

Postup vytvoření GRE tunelu je v mém případě následující:

1. vytvoření tunelového rozhraní tak, aby mohlo přijímat i odesílat pakety,
2. nastavení IP adresy na tomto rozhraní.

Tento postup odpovídá (pro lepší představu) této sadě příkazů balíku iproute2:

```
ip link add tun type gre remote 192.168.1.102 local 192.168.1.101
ip link set tun up
ip addr add 10.0.1.1/24 dev tun
```

Výpis 13: Vytvoření tunelu pomocí iproute2

kde vytvoříme rozhraní jménem „tun“, které bude typu GRE, se vzdálenou adresou konce tunelu 192.168.1.102 a lokální 192.168.1.101. Funkce dalších příkazů je zřejmá.

V mé testovací topologii (viz kapitola 7.2) není třeba přidávat záznam do směrovací tabulky, protože si jej kernel přidá sám. To by ale neplatilo pro složitější topologie, takže zde uvádím i postup, který je ekvivalentem příkazu *ip route add 10.0.2.0/24 dev tun*.

(Poznámka: Ačkoliv v manuálových stránkách – *man ip link* – není vůbec uvedeno, že by tento příkaz uměl vytvořit GRE tunel, tak jej vytvořit umí. Z toho důvodu v nich nejsou vysvětleny ani jednotlivé možnosti tohoto příkazu, které odpovídají jednotlivým atributům v odesílané zprávě. Je možno se obrátit na manuálové stránky příkazu *ip tunnel*, v němž jsou tyto možnosti popsány. Příkaz *ip tunnel* vytváří rozhraní pomocí ioctl.)

Postup pomocí Netlink socket je popsán v následujících podkapitolách.

7.1.1 Vytvoření rozhraní

Prezentovaný kód je součástí metody main(), pokud nejde o funkce.

```
struct sockaddr_nl sockdest;
struct msghdr msg;
struct iovec iov;

struct nlattr *linkinfo;
struct nlattr *infodata;

struct {
    struct nlmsghdr nlm;
    struct ifinfomsg ifi;
    char reqbuf[1024];
} req;

int fd, status;
char saddress[sizeof(struct in_addr)];
char daddress[sizeof(struct in_addr)];
char *type;
char *name;
unsigned char pmtu = 1, ttl = 0;
```

Výpis 14: Úsek programu – deklarace proměnných (ifinfomsg)

Výpis 14:
Zde jsou uvedeny všechny struktury a proměnné, které budou potřeba. Proměnná req je tvořena Netlink hlavičkou nlmsghdr, RTNetlink hlavičkou ifinfomsg a bufferem, který bude sloužit pro ukládání atributů.

```
fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
memset(&req, 0, sizeof(req));

req.nlm.nlmsg_type = RTM_NEWLINK;
req.nlm.nlmsg_flags = NLM_F_CREATE | NLM_F_REQUEST | NLM_F_EXCL;
req.nlm.nlmsg_len = NLMSG_LENGTH(sizeof(struct ifinfomsg));
req.ifi.ifi_flags |= IFF_UP;
req.ifi.ifi_change = 0xFFFFFFFF;
```

Výpis 15: Úsek programu – otevření socketu a nastavení hlaviček (ifinfomsg)

Výpis 15:
Vytvořím socket, vynuluji paměť pro *req* a nastavím hlavičky.
Netlink hlavička:

- RTM_NEWLINK – budu vytvářet rozhraní,
- NLM_F_CREATE | NLM_F_REQUEST | NLM_F_EXCL – vytvářím nový objekt, čímž posílám do kernelu požadavek, a chci, aby se nové rozhraní nevytvořilo, pokud už existuje rozhraní stejného jména nebo indexu,

- NLMSG_LENGTH(sizeof(struct ifinfomsg)) – nastavím počáteční délku zprávy jako délku nlmsghdr + ifinfomsg, z této délky pak budu vycházet při přidávání atributů;

RTNetlink hlavička:

- IFF_UP – rovnou tímto flagem nastavím, že rozhraní bude moct přijímat i odesílat pakety. Operace |= je bitový OR, který k defaultnímu flagu (samé nuly) přidá flag IFF_UP;
- 0xFFFFFFFF – nastavím masku flagů; stejně by fungovalo nastavit toto pole na stejnou hodnotu jako ifi_flags;

```
if (! inet_pton (AF_INET, argv[2], daddress))
   {  printf ("\nError converting address."); }

if (! inet_pton (AF_INET, argv[3], saddress))
   {  printf ("\nError converting address."); }

name = strdup(argv[1]);
type = strdup("gre");
```

Výpis 16: Úsek programu – definice proměnných pro atributy (ifinfomsg)

Výpis 16:
Předpokládám spouštění programu s parametry (*jméno rozhraní*, *cílová adresa*, *zdrojová adresa*). Funkce inet_pton() zkopíruje IP adresu z textové do binární formy a uloží ji do proměnné daddress, resp. saddress. AF_INET označuje, že jde o IPv4 adresy. [58] Funkce strdup() vrací pointer na zkopírovaný string. [59]

```
addAttr(&req.nlm, IFLA_IFNAME, name, strlen(name));

 linkinfo  = addNested(&req.nlm, IFLA_LINKINFO);
addAttr(&req.nlm, IFLA_INFO_KIND, type, strlen(type));

infodata  = addNested(&req.nlm, IFLA_INFO_DATA);
addAttr(&req.nlm, IFLA_GRE_LOCAL, &saddress, 4);
addAttr(&req.nlm, IFLA_GRE_REMOTE, &daddress, 4);
addAttr(&req.nlm, IFLA_GRE_PMTUDISC, &pmtu, 1);
addAttr(&req.nlm, IFLA_GRE_TTL, &ttl, 1);
```

Výpis 17: Úsek programu – přidání atributů (ifinfomsg)

Výpis 17: Přidání atributů. Funkcí addAttr se přidávají atributy jeden za druhým, funkcí addNested dochází k vytvoření vnořeného atributu. Jak jsem psal v kapitole 5.3.4, na pořadí atributů kromě těch vnořených nezáleží, takže např. atribut IFLA_GRE_LOCAL by mohl být umístěn až za atributem IFLA_GRE_TTL apod.

Jednotlivé atributy (úroveň odrážek odpovídá úrovni vnoření) znamenají:

- IFLA_IFNAME – jméno rozhraní, např. „tun“,
- IFLA_LINKINFO – vnořený atribut obsahující informaci o typu rozhraní:
  - IFLA_INFO_KIND – rozhraní bude typu *gre*,
  - IFLA_INFO_DATA – vnořený atribut obsahující další informace:
    - IFLA_GRE_LOCAL – zdrojová IP adresa tunelu, musí jít o adresu jiného rozhraní daného počítače,
    - IFLA_GRE_REMOTE – cílová IP adresa tunelu,
    - další tributy jsou volitelné, součástí kódu jso pro lepší názornost.
      IFLA_GRE_PMTUDISC – nastavením na 0 vypneme defaultně zapnutou Path MTU Discovery, což je technika zjišťující největší velikost paketu takovou, aby při cestě sítí nedocházelo k jeho fragmentaci. [63]
      IFLA_GRE_TTL – umožňuje nastavit hodnotu TTL v rozsahu 1-255, nastavením 0 (nebo vynecháním atributu) se zvolí defaultní (zděděná) hodnota, která je 64 (viz /linux/ip.h). [53]

```
#define NLA_DATA(na)  ((void *)((char*)(na) + NLA_HDRLEN))

static int addAttr(struct nlmsghdr *nlm, int attrlabel , const void *data, int datalen)
{
   struct nlattr *nla = (struct nlattr *) (((void *) nlm) + nlm->nlmsg_len);
   unsigned int attrlen = NLA_ALIGN(NLA_HDRLEN + datalen);
   nla->nla_len = attrlen;
   nla->nla_type = attrlabel;
   memcpy(NLA_DATA(nla), data, datalen);

   nlm->nlmsg_len = nlm->nlmsg_len + (NLA_HDRLEN + NLA_ALIGN(datalen));
   return 0;
}

static struct nlattr * addNested(struct nlmsghdr *nlm, int attrlabel)
{
   struct nlattr *nestattr = (struct nlattr *) ((void *) nlm + nlm->nlmsg_len);
   if (!addAttr(nlm, attrlabel , NULL, 0));
   { return nestattr ; }
}
```

Výpis 18: Úsek programu – funkce pro přidání atributů

Výpis 18:
Zde se nachází funkce připojující atributy za RTNetlink hlavičku. Funkcí addAttr se připojují obyčejné atributy a v parametrech jí je předáno (viz výpis 17): pointer na Netlink hlavičku, typ atributu, pointer na data související s daným typem atributu (např. IP adresa nebo hodnota) a délka těchto dat.

Funkce provádí následující:

- část paměti za RTNetlink hlavičkou přetypuje na nlattr. Zde u přidání prvního atributu využiji počáteční délku zprávy nastavenou ve výpisu 15, činící sizeof(nlmsghdr)+sizeof(ifinfomsg);
- spočítá délku atributu (délka nlattr hlavičky + délka dat) a zaokrouhlí ji,
- v hlavičce atributu nastaví tuto spočtenou délku,
- v hlavičce atributu nastaví typ atributu předaný parametrem,
- zkopíruje data do paměti hned za nlattr hlavičku, tedy na místo, kam ukazuje makro NLA_DATA,
- aktualizuje délku zprávy v hlavičce nlmsghdr o délku právě připojeného atributu. Bez této aktualizace by byly přidané atributy ignorovány; její špatnou aktualizací by se nepřipojovaly správně.

Funkcí addNested – v parametrech jí je předán pointer na Netlink hlavičku a typ atributu – dojde k přidání vnořeného atributu:

- stejně jako v předchozím případě funkce přetypuje volné místo za RTNetlink hlavičkou či za posledním atributem na nlattr;
  - pro složitější programy by zde asi bylo vhodné dopsat kontrolu, zda ještě nějaké volné místo zbývá;
- zavolá se funkce pro přidání obyčejného atributu. Vnořený atribut je tedy speciální typ obyčejného atributu. Nepřidají se za něj (za hlavičku) žádná data a délka zprávy v hlavičce nlmsghdr se zvýší o velikost hlavičky nlattr (4 B). Tato funkce zároveň vrací pointer na tento vnořený atribut.

```
infodata->nla_len = \
    (((void *) (&req.nlm)) + NLMSG_ALIGN((&req.nlm)->nlmsg_len)) - (void *) infodata;

linkinfo ->nla_len = \
    (((void *) (&req.nlm)) + NLMSG_ALIGN((&req.nlm)->nlmsg_len)) - (void *) linkinfo;
```

Výpis 19: Úsek programu – dokončení vnoření atributů

Výpis 19:

V kapitole 5.3.4.1 jsem psal o tom, že délka nebo stupeň zanoření atributů je dána nastavenou délkou v hlavičce vnořeného atributu (nla_len). A vnoření atributů v tomto programu jsem znázornil u výpisu 17, v němž také ukládám do proměnných *linkinfo* a *infodata* pointery na vytvářené vnořené atributy IFLA_LINKINFO, resp. IFLA_INFO_DATA. Tyto pointery zde využiji k „uzavření“ vnoření.

Když mám v paměti uložené všechny atributy (a obě vnoření končí s tímto posledním atributem), je možno délku vnoření spočítat odečtením pointerů, protože Netlink hlavička, RTNetlink hlavička i atributy představují v paměti jeden souvislý blok (viz deklarace

proměnných, výpis 14). Čili délka vnořených atributů je rovna: pointer na strukturu nlmsghdr *plus* délka zprávy, čímž získám pointer na konec zprávy (konec posledního atributu), to celé *mínus* pointer na začátek vnořených atributů. Výsledek se uloží do pole nla_len odpovídajícího atributu.

```
memset(&sockdest, 0, sizeof(sockdest));
sockdest.nl_family = AF_NETLINK;

memset(&msg, 0, sizeof(msg));
msg.msg_name = &sockdest;
msg.msg_namelen = sizeof(sockdest);

iov.iov_base = &req.nlm;
iov.iov_len = req.nlm.nlmsg_len;
msg.msg_iov = &iov;
msg.msg_iovlen = 1;

status = sendmsg(fd, &msg, 0);
if (status < 0) {
   perror("send");
    return 1; }
```

Výpis 20: Úsek programu – sestavení a odeslání zprávy

Výpis 20:

Tyto struktury byly popsány v kapitole 5.1 a tento postup v manuálových stránkách. [41] Budu odesílat data do kernelu, proto nastavím nl_family v proměnné sockdest jako AF_NETLINK, nl_pid je 0. Tímto určím cílovou adresu. Do msg_name uložím její adresu v paměti (jde o pointer) a do msg_namelen její délku.

Do iov_base uložím adresu Netlink hlavičky, do iov_len délku odesílané zprávy a do msg_iov adresu proměnné iov. Jedná se o jednu strukturu, takže msg_iovlen nastavím na 1.

Teď už zbývá zprávu jen odeslat. Parametry funkce sendmsg() jsou *otevřený socket*, *zpráva msghdr* a *flagy*. Funkce vrací velikost odeslané zprávy, případně -1 při chybě. [62]

7.1.2 Přiřazení IP adresy

Vytváření a odesílání zprávy i deklarace proměnných fungují v dalších programech analogicky, takže je už nebudu zmiňovat.

```
req.nlm.nlmsg_len = NLMSG_LENGTH(sizeof(struct ifaddrmsg));
req.nlm.nlmsg_type = RTM_NEWADDR;
req.nlm.nlmsg_flags = NLM_F_CREATE | NLM_F_REQUEST;
req.ifa.ifa_family = AF_INET;
req.ifa.ifa_prefixlen = 24;
req.ifa.ifa_index = if_nametoindex(argv[1]);
```

Výpis 21: Úsek programu – nastavení hlaviček (ifaddrmsg)

Výpis 21:

Netlink hlavička je velmi podobná té z vytváření rozhraní. Pokud přidávám IPv4 adresu, musím v RTNetlink hlavičce nastavit ifa_family na AF_INET. Dále určím počet bitů síťové části adresy (zde 24) a pak musím určit rozhraní (podle indexu), na které se adresa přidá. Předpokládám spouštění programu s parametry *jméno rozhraní*, *ip adresa*.

```
if (!inet_pton(AF_INET, argv[2], address)) {
    printf ("\nError converting address."); }

nla = (struct nlattr *) req.reqbuf;
nla->nla_type = IFA_LOCAL;
nla->nla_len = sizeof(struct nlattr) + 4;
memcpy(NLA_DATA(nla), &address, 4);

req.nlm.nlmsg_len += nla->nla_len;
```

Výpis 22: Úsek programu – přidání atributu (ifaddrmsg)

Výpis 22:

Konkrétní IPv4 adresu předám kernelu jako atribut IFA_LOCAL. Jde o jediný atribut, proto jsem si mohl dovolit vynechat funkci a přidat jej „ručně“. Do nla_len se přičítají (k nlattr hlavičce) 4 bajty, tedy velikost IPv4 adresy, a po přidání atributu je třeba upravit délku zprávy, nlmsg_len.

7.1.3 Přidání záznamu do směrovací tabulky

```
req.nlm.nlmsg_type = RTM_NEWROUTE;
req.nlm.nlmsg_flags = NLM_F_CREATE | NLM_F_REQUEST | NLM_F_EXCL;
req.rtm.rtm_family = AF_INET;
req.rtm.rtm_table = RT_TABLE_MAIN;
req.rtm.rtm_scope = RT_SCOPE_UNIVERSE;
req.rtm.rtm_protocol = RTPROT_BOOT;
req.rtm.rtm_type = RTN_UNICAST;
```

Výpis 23: Úsek programu – nastavení hlaviček (rtmsg)

Výpis 23:

Rodina adres je AF_INET; *protocol* nastavím na RTPROT_BOOT, protože cestu přidávám ručně (viz kapitola 5.3.3), nicméně je možné kernelu podstrčit i jiné nastavení. Jedná se o cestu na reálnou adresu v síti, takže *type* bude RTN_UNICAST, obvykle by tato adresa byla dále než ve stejném subnetu, takže *scope* bude RT_SCOPE_UNIVERSE a bude vložena do tabulky Main.

```
if_index = if_nametoindex(argv[1]);
req.rtm.rtm_dst_len = atoi(argv[3]);   // prevod na int

if (!inet_pton(AF_INET, argv[2], address))
   { printf ("\nError converting address."); }

addAttr(&req.nlm, RTA_DST, &address, 4);
addAttr(&req.nlm, RTA_OIF, &if_index, 4);
```

Výpis 24: Úsek programu – přidání atributů (rtmsg)

Výpis 24:

Předpokládám spouštění programu s atributy *jméno rozhraní*, *adresa cílové sítě*, *prefix cílové sítě*. Prefix sítě se musí nastavit do RTNetlink hlavičky (*rtm_dst_len*). Atributy pak jsou dva – RTA_DST pro adresu cílové sítě a RTA_OIF pro určení odchozího rozhraní pro pakety mířící do dané sítě.

Výsledkem těchto programů je vytvoření funkčního GRE tunelu, viz výpis 25. Tunelovému rozhraní se automaticky nastavil flag POINTOPOINT a MTU se snížilo o velikost přidaných hlaviček:

```
% ip add sh tun
8: tun: <POINTOPOINT,UP,LOWER_UP> mtu 1476 qdisc noqueue state UNKNOWN
    link/gre 192.168.1.101 peer 192.168.1.102
    inet 10.0.1.1/24 scope global tun
    inet6 fe80::5efe:c0a8:165/64 scope link
       valid_lft forever preferred_lft forever

% ip route sh
default via 192.168.1.1 dev wlan0 proto static
10.0.1.0/24 dev tun proto kernel scope link src 10.0.1.1
192.168.1.0/24 dev wlan0 proto kernel scope link src 192.168.1.101
```

Výpis 25: Vytvořený tunel

7.2 Ethernet over GRE

Pokud chceme v GRE přenášet ethernetové rámce, je postup takřka totožný s postupem pro IPoGRE. Jediný rozdíl je při vytváření rozhraní, kdy jako IFLA_INFO_KIND bude *gretap* místo *gre*, tedy viz výpis 26. Tento postup ostatně odpovída postupu v iproute, kdy *gre* ve výpisu 13 nahradíme *gretap*.

```
..
type = strdup("gretap");
..
addAttr(&req.nlm, IFLA_INFO_KIND, type, strlen(type));
..
```

Výpis 26: ETHERNEToGRE

8 Testovací síťová topologie

Testování vytvořených tunelů jsem prováděl v domácích podmínkách se dvěma fyzickými počítači. Topologie (obrázek 14) sice neodpovídá typickému použití tunelů, nicméně k otestování jeho funkčnosti je dostatečná.

Verze kernelu (*uname -r*) a typ síťové karty byly na obou počítačích následující:

1. 3.5.0-17-generic, Realtek Semiconductor RTL8139/8139C/8139(L)+,
2. 3.8.6-1-ARCH, Realtek Semiconductor RTL8111/8168.

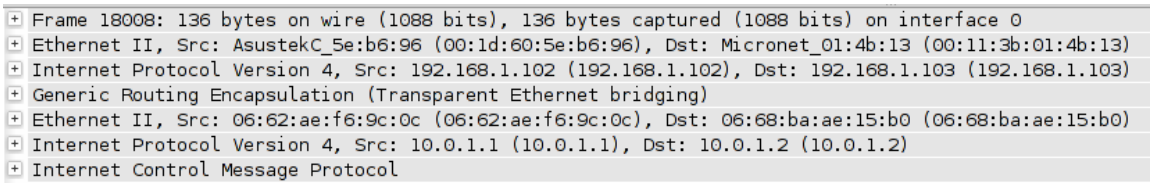
Adresy na obrou rozhraních eth0 slouží jako začátek/konec tunelu.

Obrázek 14: Topologie testovací sítě

Pro základní otestování úspěšnosti enkapsulace postačí příkaz *ping* spolu s odchycením provozu (např. programem Wireshark) a analýzou zachycených paketů. Zachycení probíhá na výstupním rozhraní (eth0) a provoz musí pocházet z tunelového rozhraní, což se díky záznamu ve směrovací tabulce děje automaticky – pro pakety směřující do sítě 10.0.1.0/24 je nastaveno jako výstupní rozhraní to tunelové. Zachycený provoz tedy v případě IPoGRE obsahuje enkapsulaci třetí vrstvy (obrázek 15) a v případě ETHERNEToGRE enkapsulaci druhé vrstvy (obrázek 16).

```
+ Frame 58: 122 bytes on wire (976 bits), 122 bytes captured (976 bits) on interface 0
+ Ethernet II, Src: AsustekC_5e:b6:96 (00:1d:60:5e:b6:96), Dst: Micronet_01:4b:13 (00:11:3b:01:4b:13)
+ Internet Protocol Version 4, Src: 192.168.1.102 (192.168.1.102), Dst: 192.168.1.103 (192.168.1.103)
+ Generic Routing Encapsulation (IP)
+ Internet Protocol Version 4, Src: 10.0.1.1 (10.0.1.1), Dst: 10.0.1.2 (10.0.1.2)
+ Internet Control Message Protocol
```

Obrázek 15: Enkapsulace třetí vrstvy



Obrázek 16: Enkapsulace druhé vrstvy

8.1 Měření zátěže

K simulaci síťového provozu jsem použil utilitu Iperf. Ta slouží k měření výkonu sítě, funguje v režimu klient/server a umožňuje klientovi odesílat na server data předem specifikovanou rychlostí. [66]

Při odesílání dat jsem měřil zátěž CPU utilitou iostat, která počítá vytížení CPU jako průměrnou hodnotu vytížení jednotlivých procesorů v multiprocesorových systémech. Údaje o vytížení čte iostat z /proc, viz kapitola 4. [67]

```
% iperf -s -u
```

Výpis 27: Iperf – spuštění serveru

```
% iperf -c 10.0.1.2 -i 5 -b 100M -t 62
```

Výpis 28: Iperf – klient

Příkaz ve výpisu 27 spustí server (*-s*) v UDP módu (*-u*) na jednom z počítačů. Příkaz ve výpisu 28 spustí na druhém počítači Iperf v režimu klienta (*-c*), připojí se na server – zde jsem testoval zátěž bez enkapsulace (IP adresa serveru bude IP adresou rozhraní Eth0) a s enkapsulací (IP adresa serveru bude IP adresou tunelového rozhraní) –, každých 5 sekund vypíše aktuální stav (*-i 5*), rychlost přenosu bude 100 Mbit/s (*-b 100M*) a odesílání poběží 62 sekund (*-t 62*).

```
% iostat -c 1
```

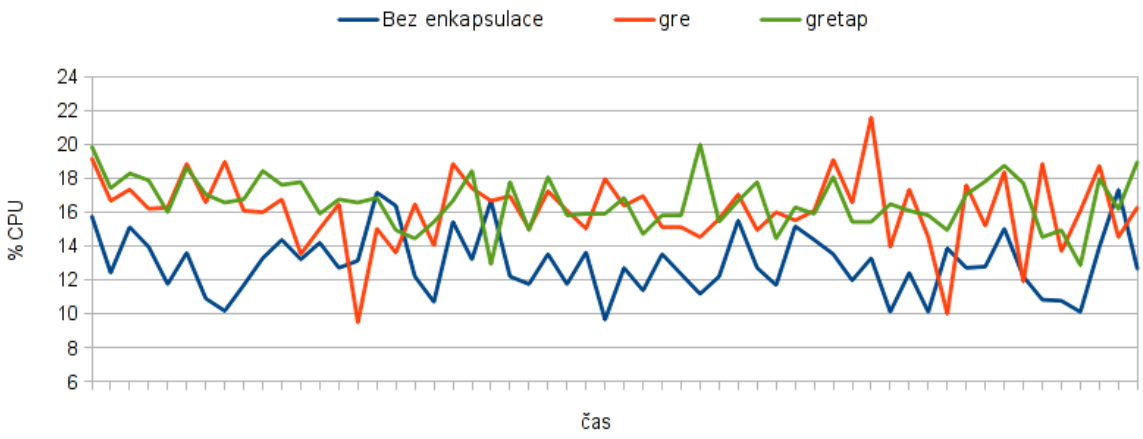
Výpis 29: Iostat – zátěž CPU

Příkazem ve výpisu 29 se spustí vypisování aktuálního vytížení CPU s intervalem 1 sekunda.

Na počítači, kde jsem měřil vytížení, neběžela žádná jiná aplikace. Osazeným CPU je AMD Turion 64 X2 TL-60 se dvěma jádry, každé je taktováno na 2 GHz.

Výsledná zátěž je zobrazena v grafu na obrázku 17. Výsledky měření jsou následující:

- bez enkapsulace
  - odesláno 707 MB dat průměrnou rychlostí 95,7 Mbit/s,
  - průměrné vytížení CPU bylo 13 %;
- GRE
  - odesláno 661 MB dat průměrnou rychlostí 89,5 Mbit/s,
  - průměrné vytížení CPU bylo 16,1 %;
- GRETAP
  - odesláno 650 MB dat průměrnou rychlostí 88 Mbit/s,
  - průměrné vytížení CPU bylo 16,6 %.



Obrázek 17: Graf zátěže pro jednotlivé enkapsulace

9 Závěr

V této práci jsem vysvětlil, co je to Netlink socket, a pak jsem důkladně popsal, jak se používá a co umožňuje. To obnášelo seznámení se se strukturami tvořícími Netlink zprávu, tedy Netlink hlavičkou, RTNetlink hlavičkami a atributy.

Stejnou Netlink hlavičku sdílí všechny zprávy, ale dle způsobu použití se liší její nastavení. RTNetlink hlaviček je více a použije se ta, která odpovídá typu prováděné operace. A na použité RTNetlink hlavičce zase závisí typy atributů, které lze použít. Po získání teoretického základu, korektním sestavení Netlink zprávy a jejím odesláním do kernelu jsem byl schopen napsat programy, které vytvoří Netlink zprávu a odešlou ji do kernelu, který použije informace v ní obsažené k vytvoření tunelového rozhraní a k přidání IPv4 adresy na rozhraní, čímž jsem vytvořil GRE tunel ve dvou verzích – jedna přenášela IP pakety a druhá ethernetové rámce. Poslední program pak přidá cestu do směrovací tabulky.

Poté jsem vytvořil jednoduchou síťovou topologii, na které jsem vytvořené tunely testoval. Porovnal jsem, jak enkapsulace ovlivňuje vytížení CPU při síťovém provozu v porovnání s provozem, jenž enkapsulován nebyl.

Vytvořené programy by se daly rozšířit tak, aby pokryly více možností Netlink socketu. Zde by patřilo třeba použití dalších dostupných RTNetlink hlaviček či rovnou použití NETLINK_FIREWALL nebo dalších protokolů místo NETLINK_ROUTE. Stejně tak jsem ve svých programech nepoužil všechny atributy, které by se za danými RTNetlink hlavičkami daly použít.

Nicméně velká část těchto v tomto textu neřešených možností je už implementována v balíku iproute, takže jako logické pokračování této práce bych spíše viděl důkladnou dokumentaci všech těchto rozšíření i s různými vlastnostmi a omezeními pro jejich použití. Za celou dobu, po kterou jsem tuto práci psal, jsem totiž žádnou ucelenou dokumentaci, jež by se tomuto problému věnovala, nenašel.

A doufám, že by tento text mohl sloužit jako návod k pokročilému použití Netlink socket, což byl ostatně i cíl, se kterým jsem jej psal.

10 Reference

[1] GERTNER, Jon. True Innovation. Innovation and the Bell Labs Miracle. *The New York Times* [online]. 25. 2. 2012, 4. 3. 2012 [cit. 2013-04-03]. Dostupné z: http://www.nytimes.com/2012/02/26/opinion/sunday/innovation-and-the-bell-labs-miracle.html?pagewanted=all&_r=1&

[2] The Creation of the UNIX Operating System: An Overview of the UNIX* Operating System. *Bell Labs* [online]. [2002] [cit. 2013-04-03]. Dostupné z: http://www.bell-labs.com/history/unix/tutorial.html

[3] JOHNSON, S. C. a B. W. KERNIGHAN. THE PROGRAMMING LANGUAGE B. *Dennis Ritchie Home Page* [online]. [1997], 29. 5. 2000 [cit. 2013-04-03]. Dostupné z: http://cm.bell-labs.com/cm/cs/who/dmr/bintro.html

[4] KERNIGHAN. Unix - Frequently Asked Questions (6/7) [Frequent posting]: Section - A very brief look at Unix history. *Internet FAQ Archives - Online Education - faqs.org* [online]. 30. 5. 1994, 8. 8. 2012 [cit. 2013-04-03]. Dostupné z: http://www.faqs.org/faqs/unix-faq/faq/part6/section-2.html

[5] CLARKE, Gavin. C and Unix pioneer Dennis Ritchie reported dead: printf("Rest in peace, Dennis\n"); exit(0);. C and Unix pioneer Dennis Ritchie reported dead. *The Register* [online]. 13. 10. 2011 [cit. 2013-04-03]. Dostupné z: http://www.theregister.co.uk/2011/10/13/dennis_ritchie/

[6] RITCHIE, Dennis. The Evolution of the Unix Time-sharing System*. Early Unix history and evolution. *Dennis Ritchie Home Page* [online]. 1984 [cit. 2013-04-03]. Dostupné z: http://cm.bell-labs.com/cm/cs/who/dmr/hist.html

[7] The Open Group Definition. *The Linux Information Project* [online]. 10. 5. 2005, 15. 3. 2006 [cit. 2013-04-03]. Dostupné z: http://www.linfo.org/open_group.html

[8] STONEBANK, M. UNIX Introduction. UNIX Tutorial - Introduction [online]. 9. 10. 2000 [cit. 2013-04-03]. Dostupné z: http://www.ee.surrey.ac.uk/Teaching/Unix/unixintro.html

[9] GOLDFELD, Arik. Everything is a File. *Unix System Administration Independent Learning* [online]. [cit. 2013-04-03]. Dostupné z: http://www.cs.bgu.ac.il/ arik/usail/concepts/filesystems/everything-is-a-file.html

[10] HANRIGOU, Philippe. In UNIX Everything is a File [online]. 2007, 2012 [cit. 2013-04-03]. Dostupné z: http://ph7spot.com/musings/in-unix-everything-is-a-file

[11] BOWER, Tim. The Hierarchical Structure of the File System. *Introduction to Unix Study Guide* [online]. [cit. 2013-04-03]. Dostupné z: http://www.sal.ksu.edu/faculty/tim/unix_sg/nonprogrammers/file_sys/file_system.html

[12] Unix Way. *Cunningham & Cunningham, Inc.* [online]. 14. 4. 2012 [cit. 2013-04-03]. Dostupné z: http://c2.com/cgi/wiki?UnixWay

[13] What is BSD?. *University Information Technology Services* [online]. 20. 7. 2012 [cit. 2013-04-03]. Dostupné z: http://kb.iu.edu/data/agom.html

[14] LEHEY, Greg. Explaining BSD. *The FreeBSD Project* [online]. 1. 10. 2012 [cit. 2013-04-03]. Dostupné z: http://www.freebsd.org/doc/en/articles/explaining-bsd/article.html

[15] Announcing breakthrough value for HP Integrity and HP-UX customers: Achieve continuous business with HP-UX, the operating system for HP mission-critical converged infrastructure. *HP Home pages worldwide* [online]. 2013 [cit. 2013-04-03]. Dostupné z: http://h71028.www7.hp.com/enterprise/w1/en/os/hpux11i-overview.html

[16] Oracle Solaris 11: The First Cloud OS. *Oracle | Hardware and Software, Engineered to Work Together* [online]. [2012] [cit. 2013-04-03]. Dostupné z: http://www.oracle.com/us/products/servers-storage/solaris/solaris11/overview/index.html

[17] STALLMAN, Richard. Initial Announcement: Free Unix!. *GNU Project - Free Software Foundation* [online]. [2012], 10. 3. 2013 [cit. 2013-04-03]. Dostupné z: http://www.gnu.org/gnu/initial-announcement.en.html

[18] Overview of the GNU System. *GNU Project - Free Software Foundation* [online]. [2012], 10. 3. 2013 [cit. 2013-04-03]. Dostupné z: http://www.gnu.org/gnu/gnu-history.en.html

[19] GNU Hurd. *GNU Project - Free Software Foundation* [online]. [2013] [cit. 2013-04-03]. Dostupné z: http://www.gnu.org/software/hurd/hurd.html

[20] Download GNU. *GNU Project - Free Software Foundation* [online]. [2013] [cit. 2013-04-03]. Dostupné z: http://www.gnu.org/software/software.en.html#allgnupkgs

[21] ROSS, Seth. A Quick History of UNIX. *Introducing the book . . . Unix System Security Tools* [online]. 1999 [cit. 2013-04-03]. Dostupné z: http://www.albion.com/security/intro-2.html

[22] CHITNIS, Atul. Why August 25th?. *Bangalore Linux User Group* [online]. [cit. 2013-04-03]. Dostupné z: http://linux-bangalore.org/blug/articles/bday.php

[23] Linux History. History of Linux, Who Invented Linux, How Was Linux Invented. *The Internet* [online]. [cit. 2013-04-03]. Dostupné z: http://www.livinginternet.com/i/iw_unix_gnulinux.htm

[24] The GNU General Public License v3.0. *GNU Project - Free Software Foundation* [online]. 2007 [cit. 2013-04-03]. Dostupné z: http://www.gnu.org/licenses/gpl.html

[25] JONES, M. Tim. Anatomy of the Linux kernel: History and architectural decomposition. *IBM developerWorks: IBM's resource for developers and IT professionals* [online]. 6. 6. 2007 [cit. 2013-04-03]. Dostupné z: http://www.ibm.com/developerworks/linux/library/l-linux-kernel/

[26] JONES, M. Tim. Kernel command using Linux system calls: Explore the SCI and add your own calls. *IBM developerWorks: IBM's resource for developers and IT professionals* [online]. 10. 1. 2010 [cit. 2013-04-26]. Dostupné z: https://www.ibm.com/developerworks/linux/library/l-system-calls/

[27] The GNU C Library: System Calls. *GNU Project - Free Software Foundation* [online]. [cit. 2013-04-26]. Dostupné z: http://www.gnu.org/software/libc/manual/html_mono/libc.html#System-Calls

[28] The Virtual Filesystem. BOVET, Daniel Pierre a Marco CESATI. *Understanding the Linux Kernel*. 3rd ed. Sebastopol: O'Reilly, 2005, xvi, 923 s. ISBN 978-0-596-00565-8.

[29] Linux kernel development by the numbers. *Pingdom Website monitoring* [online]. 16. 4. 2012 [cit. 2013-04-03]. Dostupné z: http://royal.pingdom.com/2012/04/16/linux-kernel-development-numbers/

[30] BALSA, Andrew D. The linux-kernel mailing list FAQ: Programming Religion. [online]. 17.10.2009 [cit. 2013-04-27]. Dostupné z: http://www.tux.org/lkml/#ss15

[31] Unix-like Definition. *The Linux Information Project* [online]. 19. 4. 2005, 18. 6. 2006 [cit. 2013-04-03]. Dostupné z: http://www.linfo.org/unix-like.html

[32] User-Space-to-Kernel Interface. BENVENUTI, Christian. *Understanding Linux network internals*. Sebastapol, CA: O'Reilly, 2006. ISBN 978-0-596-00255-8.

[33] HE, Kevin Kaichuan. Kernel Korner - Why and How to Use Netlink Socket. *Linux Journal* [online]. 5. 1. 2005 [cit. 2013-04-03]. Dostupné z: http://www.linuxjournal.com/article/7356?page=0,0

[34] GRAF, Thomas. Netlink Protocol Library Suite (libnl). [online]. 2013 [cit. 2013-04-09]. Dostupné z: http://www.infradead.org/ tgr/libnl/

[35] How the Linux kernel works. *TuxRadar Linux* [online]. 15. 3. 2009 [cit. 2013-04-03]. Dostupné z: http://www.tuxradar.com/content/how-linux-kernel-works

[36] BOWDEN, Terrehon, Bodo BAUER, Jorge NERIN, Shen FENG a Stefani SEIBOLD. THE /proc FILESYSTEM [online]. 1999, 2009 [cit. 2013-04-04]. Dostupné z: https://www.kernel.org/doc/Documentation/filesystems/proc.txt

[37] Sockets Tutorial. *Linux Howtos* [online]. [cit. 2013-04-03]. Dostupné z: http://www.linuxhowtos.org/C_C++/socket.htm

[38] JONES, M. *GNU/Linux application programming*. 2nd ed. Hingham, Mass.: Charles River Media, 2008, 667 p. ISBN 15-845-0568-0.

[39] Socket(2). *Linux man-pages online* [online]. 19. 1. 2009 [cit. 2013-04-03]. Dostupné z: http://man7.org/linux/man-pages/man2/socket.2.html

[40] HALL, Brian. System Calls or Bust. *Beej's Guide to Network Programming* [online]. 3. 6. 2012 [cit. 2013-04-03]. Dostupné z: http://beej.us/guide/bgnet/output/html/multipage/syscalls.html

[41] Netlink(7). *Linux man-pages online* [online]. 15. 3. 2013 [cit. 2013-04-03]. Dostupné z: http://man7.org/linux/man-pages/man7/netlink.7.html

[42] BOTOŠ, Csaba. Vše o iptables: úvod. *Root.cz - informace nejen ze světa Linuxu* [online]. 10. 1. 2006 [cit. 2013-04-03]. Dostupné z: http://www.root.cz/clanky/vse-o-iptables-uvod/

[43] HORMAN, Neil. RED HAT, Inc. *Understanding And Programming With Netlink Sockets* [online]. 6. 12. 2004 [cit. 2013-04-06]. Dostupné z: http://people.redhat.com/nhorman/papers/netlink.pdf

[44] MOORE, Paul. Generic Netlink HOW-TO based on Jamal's original doc [online]. 10. 11. 2006 [cit. 2013-04-03]. Dostupné z: http://lwn.net/Articles/208755/

[45] HALL, Brian. Using Internet Sockets. *Beej's Guide to Network Programming* [online]. 3. 6. 2012 [cit. 2013-04-03]. Dostupné z: http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html#theory

[46] AYUSO, Pablo Neira, Rafael M. GASCA a Laurent LEFEVRE. *Communicating between the kernel and user-space in Linux using Netlink sockets. Software: Practice and Experience* [online]. John Wiley & Sons, Ltd., 2010 [cit. 2013-04-08]. Dostupné z: http://1984.lsi.us.es/ pablo/docs/spae.pdf

[47] GARG, Manu. Sysenter Based System Call Mechanism in Linux 2.6. *manu's blog* [online]. 2006 [cit. 2013-04-03]. Dostupné z: http://articles.manugarg.com/systemcallinlinux2_6.html

[48] GRAF, Thomas. Netlink Protocol Library Suite (libnl). [online]. 9. 5. 2011 [cit. 2013-04-09]. Dostupné z: http://www.infradead.org/ tgr/libnl/doc/core.html

[49] rtnetlink(7). *Linux man-pages online* [online]. 5. 3. 2013 [cit. 2013-04-03]. Dostupné z: http://man7.org/linux/man-pages/man7/rtnetlink.7.html

[50] netlink(3). *Linux man-pages online* [online]. 5. 8. 2012 [cit. 2013-04-03]. Dostupné z: http://man7.org/linux/man-pages/man3/netlink.3.html

[51] GRAF, Thomas. Routing Family Netlink Library (libnl-route). [online]. 11. 8. 2011 [cit. 2013-04-10]. Dostupné z: http://www.infradead.org/ tgr/libnl/doc/route.html

[52] netdevice(7). *Linux man-pages online* [online]. 26. 4. 2012 [cit. 2013-04-03]. Dostupné z: http://man7.org/linux/man-pages/man7/netdevice.7.html

[53] PAKTRONIX SYSTEMS, LLC. IPROUTE2 Utility Suite Howto. *Policy Routing* [online]. 2001 [cit. 2013-04-10]. Dostupné z: http://www.policyrouting.org/iproute2.doc.html

[54] rtnetlink(3). *Linux man-pages online* [online]. 24. 3. 2012 [cit. 2013-04-03]. Dostupné z: http://man7.org/linux/man-pages/man3/rtnetlink.3.html

[55] VAITTINEN, Matti. Netlink sockets. *Programmer's diary* [online]. 14. 9. 2011 [cit. 2013-04-16]. Dostupné z: http://maz-programmersdiary.blogspot.cz/2011/09/netlink-sockets.html

[56] SALIM, J., H. KHOSRAVI, A. KLEEN a A. KUZNETSOV. RFC 3549 - Linux Netlink as an IP Services Protocol. *The Internet Engineering Task Force (IETF)* [online]. 2003 [cit. 2013-04-16]. Dostupné z: http://tools.ietf.org/html/rfc3549

[57] Iproute2. *The Linux Foundation* [online]. 19. 11. 2009 [cit. 2013-04-16]. Dostupné z: http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2

[58] inet_pton(3). *Linux man-pages online* [online]. 18. 6. 2008 [cit. 2013-04-18]. Dostupné z: http://man7.org/linux/man-pages/man3/inet_pton.3.html

[59] strdup(3). *Linux man-pages online* [online]. 10. 5. 2012 [cit. 2013-04-18]. Dostupné z: http://man7.org/linux/man-pages/man3/strdup.3.html

[60] CONVI, Marco. Netlink (2.6.8). [online]. 2005 [cit. 2013-04-19]. Dostupné z: http://www.oocities.org/marco_corvi/games/lkpe/netlink/netlink.htm

[61] UDUGAMA, Asanga. Manipulating the Networking Environment Using RT-NETLINK. *Linux Journal* [online]. 30. 3. 2006 [cit. 2013-04-19]. Dostupné z: http://www.linuxjournal.com/article/8498

[62] send(3). *Linux man-pages online* [online]. 23. 4. 2012 [cit. 2013-04-18]. Dostupné z: http://man7.org/linux/man-pages/man2/send.2.html

[63] Internet Protocol Version 4 (IPv4): Concepts. BENVENUTI, Christian. *Understanding Linux network internals*. Sebastapol, CA: O'Reilly, 2006. ISBN 978-0-596-00255-8.

[64] Routing: Concepts. BENVENUTI, Christian. *Understanding Linux network internals*. Sebastapol, CA: O'Reilly, 2006. ISBN 978-0-596-00255-8.

[65] Routing: Miscellaneous Topics. BENVENUTI, Christian. *Understanding Linux network internals*. Sebastapol, CA: O'Reilly, 2006. ISBN 978-0-596-00255-8.

[66] *Iperf - The TCP/UDP Bandwidth Measurement Tool* [online]. [cit. 2013-04-24]. Dostupné z: http://iperf.fr/

[67] iostat(1). *Linux man page* [online]. [cit. 2013-04-18]. Dostupné z: http://linux.die.net/man/1/iostat

[68] HANKS, S., T. LI, D. FARINACCI a P. TRAINA. Generic Routing Encapsulation (GRE). *The Internet Engineering Task Force (IETF)* [online]. 1994 [cit. 2013-04-26]. Dostupné z: http://tools.ietf.org/html/rfc1701

A Přílohy na CD

Na přiloženém CD se nacházejí programy vytvořené pro tuto práci a jejich zdrojové kódy. Obsahem CD konkrétně je:

- setLink.c – zdrojový kód pro vytvoření rozhraní,
- setLink – spustitelný soubor,
- setAddress.c – zdrojový kód pro přidání IP adresy,
- setAddress – spustitelný soubor,
- setRoute.c – zdrojový kód pro přidání cesty do směrovací tabulky,
- setRoute – spustitelný soubor,
- createGRE – skript, který zjistí aktuální IP adresu, kterou předá jako parametr pro vytvoření tunelu (lokální adresa) a spouští předchozí soubory.